Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências e Tecnologia - Campus de Presidente Prudente

LOBUS-OWL: Modelo de Localização de Defeitos em Código-Fonte Apoiado por Ontologia

ALISSON SOLITTO DA SILVA

FCT-Unesp – Presidente Prudente
Julho/2022

LOBUS-OWL: Modelo de Localização de Defeitos em Código-Fonte Apoiado por Ontologia

ALISSON SOLITTO DA SILVA

Orientador: *Dr. Rogério Eduardo Garcia*

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação, da Universidade Estadual Paulista (UNESP) - Campus de Presidente Prudente, como requisito para obtenção do título de Mestre em Ciência da Computação.
Área de Concentração: Computação Aplicada
Linha de Pesquisa: Sistemas de Informação

FCT-Unesp – Presidente Prudente
Julho/2022

S586l

Silva, Alisson Solitto da
LOBUS-OWL : Modelo de Localização de Defeitos em Código-Fonte Apoiado por Ontologia / Alisson Solitto da Silva. -- Presidente Prudente, 2022
99 p. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências e Tecnologia, Presidente Prudente
Orientador: Rogério Eduardo Garcia

1. Localização de Defeitos. 2. Ontologia. 3. Manutenção Corretiva. 4. Engenharia de Software. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências e Tecnologia, Presidente Prudente. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Presidente Prudente

CERTIFICADO DE APROVAÇÃO

TÍTULO DA DISSERTAÇÃO: LOBUS-OWL: Modelo de Localização de Defeitos em Código-Fonte Apoiado por Ontologia

AUTOR: ALISSON SOLITTO DA SILVA
ORIENTADOR: ROGÉRIO EDUARDO GARCIA

Aprovado como parte das exigências para obtenção do Título de Mestre em Ciência da Computação, área: Computação Aplicada pela Comissão Examinadora:

Prof. Dr. ROGÉRIO EDUARDO GARCIA (Participaçao Virtual)
Departamento de Matemática e Computação / UNESP/Câmpus de Presidente Prudente

Prof. Dr. LEONARDO CASTRO BOTEGA (Participaçao Virtual)
Programa de Pós-Graduação em Ciência da Informação / Unesp, Faculdade de Filosofia e Ciências, Marília

Prof. Dr. MARCELO MEDEIROS ELER (Participaçao Virtual)
Escola de Artes, Ciências e Humanidades / Universidade de São Paulo

Presidente Prudente, 25 de julho de 2022

Este trabalho é dedicado a Deus, a minha família e a minha esposa. A conclusão deste trabalho e de mais uma etapa acadêmica se deu graças a todo o esforço e dedicação dos meus pais com a minha educação desde cedo.

Agradecimentos

Agradeço primeiramente a **Deus**, por sempre ter iluminado o meu caminho e por sempre me conceder saúde para viver e seguir com os meus objetivos.

A minha família e a minha esposa, por todo apoio, carinho e compreensão.

Ao orientador e coorientador do presente trabalho, que confiaram em mim e me incentivaram para o desenvolvimento deste.

A todos os professores do curso pela dedicação e comprometimento ao transmitir conhecimento durante as disciplinas.

A todos os amigos e professores do curso de Ciência da Computação (2014-2018) e aos meus amigos do Departamento de Sistemas de Informação (2017-2020) do UNIVEM que me apoiaram a seguir nessa jornada.

Agradeço a todos os professores ao longo de toda a minha vida, que direta ou indiretamente contribuíram para minha formação pessoal e profissional. Os professores são essenciais para a evolução da sociedade!

Porque as pessoas que são loucas o suficiente para achar que podem mudar o mundo, são as que o fazem.
Steve Jobs

Resumo

Os relatórios de defeitos ajudam a triagem de inconsistências encontradas durante todo o ciclo de vida do software, por possuírem informações valiosas para a Engenharia de Software. O processo de localização de defeitos busca identificar artefatos de código-fonte possivelmente relacionados ao defeito reportado. Essa tarefa executada de maneira manual é onerosa para os programadores responsáveis pela correção, que geralmente precisam realizar a análise no código-fonte para identificar o artefato defeituoso e, depois, realizar as manutenções necessárias. O processo de localização de defeitos auxilia os programadores delimitando o espaço de busca e identificando arquivos relevantes relacionados ao defeito. Geralmente, as metodologias adotadas para o processo de localização de defeitos utilizam técnicas de aprendizado de máquina com similaridade textual, algoritmos de classificação e agrupamento de arquivos de código-fonte de acordo com os dados extraídos dos relatórios de defeitos. Nessas abordagens de localização de defeitos, os arquivos de código-fonte são considerados como unidades, podendo gerar ruídos quando o arquivo de código-fonte é grande. Neste trabalho é apresentado um modelo de localização de defeitos utilizando o conhecimento semântico do código-fonte com o apoio de ontologias. O desempenho do modelo proposto foi avaliado em seis projetos de software relevantes de código aberto (AutoMapper, MsBuild, EfCore, AspNetCore, MQTTnet e NLog) utilizando as métricas de avaliação *Top N Rank of Files* (TNRF), *Mean Reciprocal Rank* (MRR) e *Mean Average Precision* (MAP). Os resultados mostram que o modelo proposto alcança resultados significativos.

Palavras-chave: Localização de Defeitos, Ontologia, Manutenção Corretiva, Engenharia de Software.

Abstract

Bug reports help triage inconsistencies found throughout the software lifecycle, as these reports contain valuable information for Software Engineering. The bug location process seeks to identify source code artifacts possibly related to the reported bug. This manually performed task is costly for the programmers responsible for the correction, who usually need to perform analysis on the source code to identify the defective artifact and then carry out the necessary maintenance. The bug location process helps programmers by delimiting the search space and identifying relevant files related to the bug report. Generally, the methodologies adopted for the process of localization bugs use machine learning techniques with textual similarity, classification algorithms and grouping of source code files according to the data extracted from bug reports. In these bug location approaches, source code files are considered as single units, and may generate noise when the source code file is large. In this work, a bug location model is presented using semantic domain knowledge of the source code with the support of ontologies. We evaluated the performance of the proposed model in six relevant open source software projects (AutoMapper, MsBuild, EfCore, AspNetCore, MQTTnet and NLog) and carried out the experiments using the evaluation metrics Top N Rank of Files (TNRF), Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP). The results show that the proposed model achieves significant results.

Keywords: Bug Location, Ontology, Corrective Maintenance, Software Engineering.

Lista de Figuras

2.1 Parte da ontologia FOAF 21
2.2 Classificação da ontologia 23
2.3 Representação gráfica de uma tripla RDF 25
2.4 Representação gráfica tripla RDF com valor não literal 26
2.5 Funcionamento das cláusulas SELECT e WHERE em uma consulta SPARQL 31
2.6 Classificação do uso de ontologias na Engenharia de Software. Adaptado de (Happel e Seedorf, 2006) 34

3.1 Engano x Defeito x Falha x Erro 37
3.2 Relatório de defeito do sistema Bugzilla 38
3.3 Ciclo de vida de um defeito adaptado do sistema Bugzilla 41
3.4 Relatório de defeito na plataforma GitHub 43
3.5 Ciclo de vida de um defeito do projeto Botframework Solutions . . 45

4.1 Arquitetura da abordagem utilizada por Witte et al. (2007). . . . 48
4.2 Visão geral das ontologias propostas por (Kiefer et al., 2007) . . . 51
4.3 Ontologia de defeito proposta por (Tran e Le, 2014) 52
4.4 Quantidade de defeitos extraídos utilizando a ontologia proposta por (Tran e Le, 2014) 53
4.5 Precisão de pesquisa de defeitos idênticos x semelhantes (Tran e Le, 2014) 54
4.6 Arquitetura proposta para geração e uso da ontologia. (EkramiFard e Kahani, 2015) 55
4.7 Resultado de detecção comparado com a ferramenta FindBugs. (EkramiFard e Kahani, 2015) 55
4.8 Consulta SPARQL para detecação do padrão SecuritySer. (EkramiFard e Kahani, 2015) 55
4.9 Processo de serialização do código-fonte em triplas RDF. (Atzeni e Atzori, 2017) 56

4.10 Serialização RDF produzida para o método de exemplo. (Atzeni e Atzori, 2017) . . . 57
4.11 Tempo para o processo de serialização do código-fonte em 20 projetos diferentes. (Atzeni e Atzori, 2017) . . . 58
4.12 *Core Module* da ontologia OOC-O (Aguiar et al., 2019) . . . 60
4.13 Arquitetura do modelo para localização de defeitos. (Gharibi et al., 2018) . . . 61
4.14 Arquitetura do modelo para localização de defeitos proposta. (Swe e Oo, 2018) . . . 62

5.1 Acrônimo LOBUS-OWL . . . 67
5.2 Arquitetura do modelo para localização de defeitos. . . . 68
5.3 Módulo gerador de domínio semântico do código-fonte. . . . 69
5.4 Novas entidades da ontologia OOC-O em destaque. . . . 70
5.5 AST referente a declaração do método *Validate* do projeto AutoMapper . . . 71
5.6 Instância na ontologia referente a declaração do método *Validate* do projeto AutoMapper . . . 72
5.7 Módulo gerador de base de conhecimento. . . . 73
5.8 Módulo reconhecimento de entidades. . . . 75
5.9 Trecho de um relatório de defeitos. . . . 77
5.10 Exemplo de reconhecimento de entidades. . . . 78
5.11 Módulo classificação semântica de artefatos. . . . 79

6.1 Solução para obter o conjunto de dados para o experimento. . . . 83
6.2 Diagrama ER dos documentos do conjunto de dados. . . . 84
6.3 Desempenho TNRF . . . 88
6.4 Desempenho MRR e MAP . . . 88

Lista de Tabelas

4.1 Comparação das principais características dos trabalhos relacionados 66

6.1 Projetos do conjunto de dados utilizados no trabalho 85
6.2 Métricas gerador de domínio semântico do código-fonte 87
6.3 Desempenho de identificação e classificação de artefatos candidatos 87

Sumário

1 Introdução **15**
1.1 Contextualização, Justificativa e Motivação 15
1.2 Formulação do Problema 16
1.3 Objetivos Gerais e Específicos 17
1.4 Organização da Monografia 18

2 Modelagem Semântica **19**
2.1 Ontologia 19
2.2 Extensible Markup Language (XML) 23
2.3 Resource Description Framework (RDF) 24
2.4 RDF Schema (RDF-S) 27
2.5 Ontology Web Language (OWL) 28
2.6 Protocol and RDF Query Language (SPARQL) 29
2.7 Uso de Ontologias na Engenharia de Software 32
2.8 Considerações Finais 35

3 Rastreamento de Defeitos **36**
3.1 Definição de Defeitos 36
3.2 Sistemas de Rastreamento de Defeitos 37
3.2.1 Bugzilla 38
3.2.2 GitHub 42
3.3 Considerações Finais 46

4 Trabalhos Relacionados **47**
4.1 Manutenção de Software com o Apoio de Ontologias 47
4.2 Ontologia do Código-Fonte 54
4.3 Processo de Localização de Defeitos 61
4.4 Considerações Finais 63

5 LOBUS-OWL **67**
5.1 Metodologia 67

5.2 Gerador de domínio semântico do código-fonte 68
5.3 Gerador de base de conhecimento 72
5.4 Reconhecimento de entidades 74
5.5 Classificação semântica de artefatos 79

6 Prova de Conceito **82**
6.1 Conjunto de dados 82
6.2 Métricas de avaliação 85
6.3 Resultados experimentais 86

7 Considerações Finais **90**
7.1 Principais Contribuições 91
7.2 Trabalhos Futuros 92

Referências Bibliográficas **94**

CAPÍTULO 1

Introdução

1.1 Contextualização, Justificativa e Motivação

Independente da complexidade e domínio de aplicação, o software continua a evoluir ao longo do tempo. Portanto, melhorias contínuas, manutenções corretivas (em consequência de defeitos encontrados nos artefatos) e a necessidade de refatoração do código-fonte fazem parte do ciclo de vida do software.

Os defeitos podem estar presentes desde o início da concepção do software. Por exemplo, na especificação dos requisitos do sistema, na modelagem de diagramas e casos de uso, como também nos estágios finais do processo ou na implantação do sistema (Delamaro et al., 2013).

O processo de manutenção corretiva tem como objetivo localizar e sanar defeitos. Esse processo exige um esforço manual, em que o responsável deve interpretar o relato do defeito e identificar quais entidades do código-fonte devem ser alteradas para reparar o problema (Gujral et al., 2015).

Nas etapas do processo de manutenção corretiva, o processo de localização de defeitos é iniciado após um relato de defeito. A equipe responsável pelo processo de manutenção corretiva inicia um trabalho manual de interpretação do defeito reportado e busca identificar trechos do código-fonte que podem ter originado o problema. Após a identificação do artefato de código-fonte que possui o defeito, um programador é responsável por realizar as manutenções necessárias (Jeong et al., 2009).

Para apoiar as atividades de manutenção corretiva ferramentas denominadas *Bug Tracking System* (BTS) possibilitam reportar as inconsistências encontradas durante todo o ciclo de vida do software (Zhang e Lee, 2011). Com a utilização de BTS é possível armazenar e gerenciar os relatórios de defeitos (*Bugs Report*). O relatório de defeito descreve as situações em que o software

não se comporta como o esperado, contribuindo para a triagem de defeitos (Zibran, 2016).

A representação semântica do código-fonte com o uso de ontologias pode contribuir para a evolução do processo de localização de defeitos. A ontologia fornece uma camada de integração e interoperabilidade, unificando a representação de diferentes domínios em um modelo semântico (Happel e Seedorf, 2006).

A literatura registra poucos modelos que utilizam ontologias em processos relacionados à Engenharia de Software. Além disto, alguns dos modelos não descrevem uma ontologia de domínio rica. Geralmente, são estruturas que representam apenas uma taxionomia, não possuindo conceitos, nem relações, ou sequer a possibilidade de interferência de novos conhecimentos como uma ontologia.

Os trabalhos de Kiefer et al. (2007) e Tran e Le (2014) fazem uso de ontologias para a representação de artefatos do software de modo superficial, ou seja, as ontologias utilizadas representam apenas uma taxionomia de alguns elementos do domínio, sem regras e relações expressivas.

A utilização de ontologia no processo de localização de defeitos contribui para a descoberta de arquivos de código-fonte relacionados com o relato de defeito que não estão explicitamente relacionados. A ontologia permite a formalização da estrutura semântica do código-fonte, possibilitando a compreensão de seus conceitos e relações, além da inferência de novos conhecimentos.

Este trabalho propõe um modelo para a localização de defeitos em código-fonte apoiado por ontologia. O uso de ontologia possibilita a representação semântica do domínio do paradigma de programação orientado a objetos. O modelo contempla a geração de conhecimento semântico do código-fonte, até a identificação e classificação dos artefatos candidatos de código-fonte que estão relacionados com o relatório de defeito. Desta forma, é possível realizar a identificação de artefatos do código-fonte a partir de um domínio conhecido, não havendo a necessidade de utilizar abordagens de aprendizado de máquina com inúmeros dados para treinamento.

1.2 Formulação do Problema

O processo de localização de defeitos busca identificar artefatos de código-fonte candidatos relacionados ao defeito reportado. Quando essa tarefa é executada de maneira manual, torna-se onerosa para os responsáveis, pois é necessário identificar e realizar as manutenções necessárias no artefato.

As metodologias adotadas para o processo de localização de defeitos que têm como objetivo localizar artefatos candidatos a partir de um relatório de defeito geralmente utilizam técnicas de aprendizado de máquina, que consis-

tem em similaridade textual, algoritmos de classificação e agrupamento de arquivos de código-fonte de acordo com os dados extraídos dos relatórios de defeitos [Gharibi et al. (2018), Swe e Oo (2018), Chen et al. (2019), Loyola et al. (2018), Lam et al. (2017) e Rahman e Roy (2018)].

Existem duas principais abordagens no estado da arte para o processo de localização de defeitos que consideram relatos de defeitos para a recuperação da informação (Loyola et al., 2018). A primeira abordagem é baseada em similaridade, que consiste em classificar os relatórios de defeitos e os arquivos de código-fonte, obtendo como resultado os arquivos de código-fonte relacionados com o relatório de defeito de acordo o nível de similaridade entre ambos. A segunda abordagem é baseada em técnicas de aprendizado de máquina, que utilizam um conjunto de artefatos para treinamento (relatórios de defeitos históricos e seus respectivos arquivos de código-fonte defeituosos). Esta também utiliza algoritmos para classificar e agrupar documentos em um rótulo relevante por similaridade textual, resultando em uma função de pontuação para determinar os arquivos de código-fonte candidatos para correção.

Nos projetos de Gharibi et al. (2018) e Swe e Oo (2018) referentes a modelos de localização de defeitos, existem lacunas que serão exploradas nessa proposta. Uma das principais lacunas nos modelos está relacionada aos processos utilizados para obter a relação entre os artefatos de código-fonte e os relatórios de defeitos. Nenhum dos trabalhos busca representar semanticamente a estrutura do código-fonte o que aqui se questiona, uma vez que a utilização de ontologias torna possível a interoperabilidade da representação de código-fonte em diferentes linguagens. Além disso, a adição da semântica pode resultar na recuperação da informação relacionada as descrições dos relatórios de defeitos e na desambiguação de entidades identificadas nos relatos de defeitos.

As metodologias de localização de defeitos atuais consideram apenas a análise estática dos arquivos de código-fonte, não observando sua estrutura arquitetônica e semântica.

As abordagens com métodos estáticos que tratam os relatórios de defeitos como uma coleção de palavras não consideram o contexto e não exploram totalmente as informações semânticas contidas nos relatórios de defeitos

Desse modo, levanta-se o problema de como considerar a estrutura arquitetônica e semântica do código-fonte para apoiar o processo de localização de defeitos.

1.3 Objetivos Gerais e Específicos

O presente trabalho propõe um modelo de localização de defeitos com o apoio de ontologia para a representação formal do conhecimento do paradigma de

programação orientado a objetos.

Busca-se realizar a identificação e classificação de artefatos de código-fonte a partir de relatórios de defeitos considerando a estrutura semântica do código-fonte.

A fim de alcançar o objetivo principal, logo uma série de objetivos específicos são definidos:

- Desenvolver um gerador semântico do código-fonte utilizando uma ontologia para modelar o conhecimento do paradigma de programação orientada a objeto.
- Analisar e converter tokens extraídos do código-fonte em conceitos e relações da ontologia do paradigma de programação orientada a objeto.
- Extrair conceitos e instâncias da ontologia para geração de uma base de conhecimento.
- Extrair entidades encontradas no relatório de defeito de acordo com os conceitos do código-fonte.
- Identificar e classificar os artefatos de código-fonte candidatos a manutenção com base no reconhecimento de conceitos extraídos do relatório de defeito.
- Avaliar o modelo de localização de defeitos proposto analisando a eficácia da localização e classificação dos artefatos de código-fonte defeituosos.

1.4 Organização da Monografia

O presente trabalho é composto por 7 capítulos organizados na seguinte estrutura:

- Capítulo 2 fornece a base teórica de conceitos como: Web Semântica, Ontologia, SPARQL e a relação da Web Semântica com a área de Engenharia de Software;
- Capítulo 3 aborda conceitos sobre defeitos de software e sistemas de rastreamentos de defeitos;
- Capítulo 4 são apresentados os trabalhos correlatos e as discussões que envolvem esta proposta;
- Capítulo 5 tem como objetivo apresentar ao leitor a proposta de pesquisa e metodologia para o desenvolvimento do projeto de pesquisa;
- Capítulo 6 detalha o conjunto de dados e as métricas utilizadas para aferir o desempenho do trabalho.
- Capítulo 7 são apresentadas as considerações finais e os trabalhos futuros desta proposta.

CAPÍTULO 2

Modelagem Semântica

O presente capítulo apresenta conceitos utilizados neste trabalho. São discutidos os conceitos e tecnologias que envolvem o uso de ontologias. Posteriormente, é explorado o uso de ontologias na Engenharia de Software. Por último, são realizadas as considerações finais do capítulo.

2.1 Ontologia

Ontologia, na Filosofia, é a ciência do que é, dos tipos de estruturas dos objetos, propriedades, eventos e relacionamentos em todas as áreas da realidade, estabelecendo sistematicamente sua linhagem conceitual (Breitman, 2000). A definição de ontologia mais frequentemente encontrada na literatura é proposta por Gruber (1995): "Uma ontologia é uma especificação formal e explícita de uma conceitualização compartilhada".

> Aqui conceitualização representa um modelo abstrato de algum fenômeno que identifica os conceitos relevantes para o mesmo. Explícita significa que os elementos e suas restrições estão claramente definidos; formal significa que a ontologia deve ser passível de processamento automático, e compartilhada reflete a noção de que uma ontologia captura conhecimento consensual, aceito por um grupo de pessoas. (Breitman, 2000).

No entanto, na Ciência da Computação, o significado e a finalidade do termo são distintos. A ontologia na Ciência da Computação é um conjunto de informações e dados obtidos, que podem ser definidos como um conjunto de conceitos fundamentais e suas relações existentes no domínio em questão. A ontologia representa o entendimento do mundo-alvo de maneira formal e

compreensível por humanos e computadores, de forma semelhante à definida na Filosofia (Mizoguchi, 2004).

Guarino et al. (2009) complementam a visão de ontologia na Ciência da Computação, explicando que as ontologias computacionais são um mecanismo de modelar formalmente a estrutura de um sistema, descrevendo as entidades e relações relevantes do domínio em questão.

As ontologias apresentam um modelo para a descrição de dados e metadados onde é possível representar de maneira formal e semântica um determinado domínio. O principal objetivo do uso de ontologias é obter um entendimento comum de um domínio especifico, além de alcançar um entendimento comum do vocabulário do domínio, sendo um fator para o apoio à interoperabilidade semântica entre diversas aplicações.

De acordo com Isotani e Bittencourt (2015) uma ontologia pode ser formalmente definida em um conjunto contendo quatro elementos, sendo:

1. Classes que representam os conceitos fundamentais do domínio de interesse. As classes representam coisas físicas ou conceituais do domínio e podem conter subclasses para a especialização de conceitos.

2. Relações ou associações que definem formalmente as relações entre os conceitos e termos do domínio de interesse.

3. Instâncias que são representações derivadas das classes ou conceitos representados em uma ontologia.

4. Axiomas do domínio que são essenciais para definir regras e restrições inerentes às instâncias de uma ontologia.

A estrutura principal de uma ontologia consiste em uma generalização ou especialização hierárquica de conceitos, ou seja, uma taxonomia. Por exemplo, em um domínio, onde existam as entidades “Pessoa”, “Gerente” e “Pesquisador”, a entidade “Pessoa” é um conceito base para as demais. Deste modo, uma pessoa concreta (Gerente ou Pesquisador) deve ser uma instância do seu conceito correspondente.

Na Figura 2.1 é apresentada parte da ontologia FOAF (Dan Brickley, 2014). A ontologia FOAF representa o domínio de informações relacionados com as pessoas, possuindo atualmente 22 classes e 67 propriedades. As formas em círculos representam as classes da ontologia; os retângulos azuis representam as propriedades de objetos (relacionam as classes); os retângulos verdes as propriedades de dados e em amarelo seu respectivo tipo.

Stuckenschmidt e Van Harmelen (2005) destacam quatro áreas que podem ser beneficiadas com o uso de ontologias.

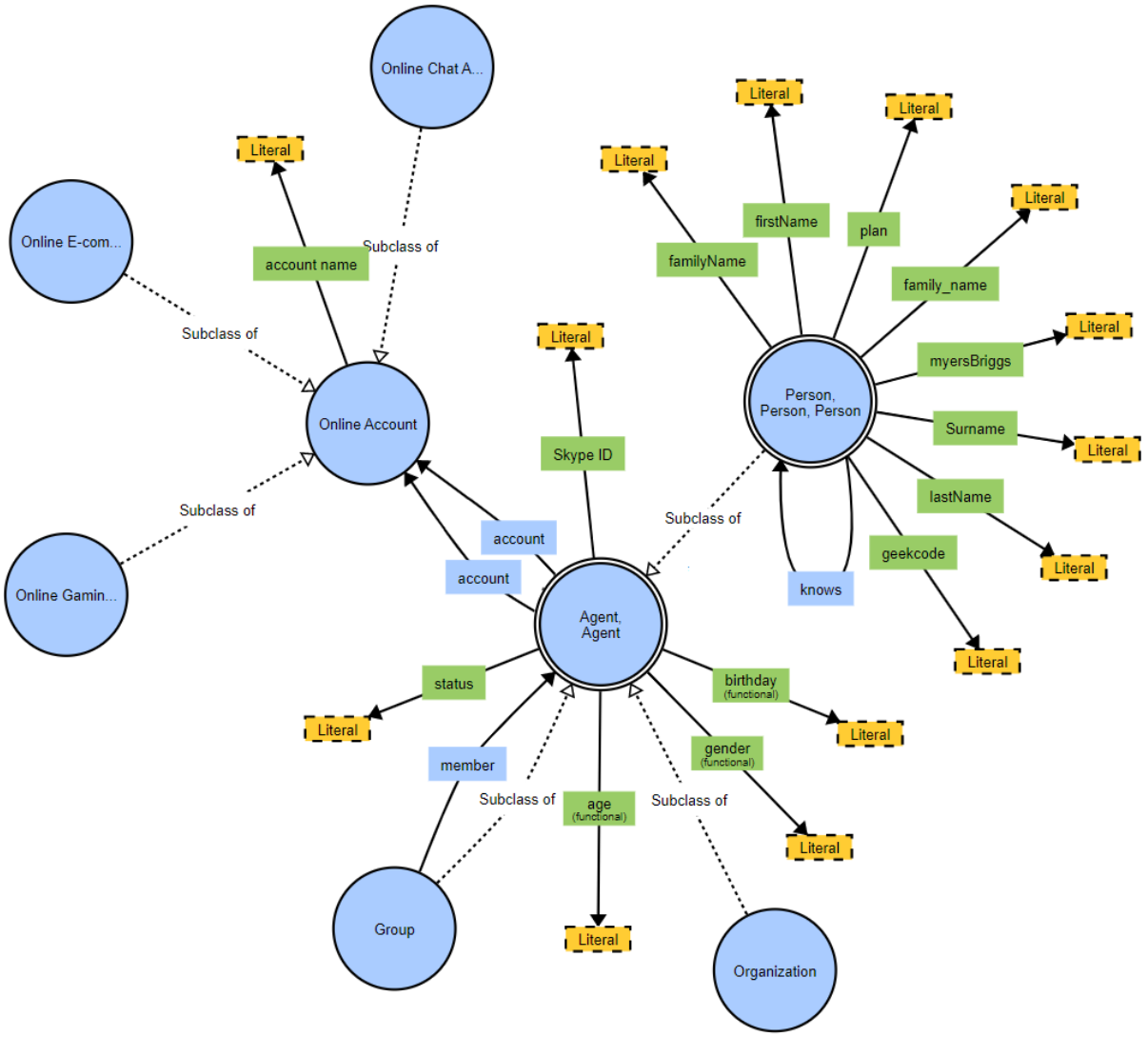


Figura 2.1: Parte da ontologia FOAF (Elaborado pelo autor).

- Comunicação: as ontologias podem ser úteis para a comunicação e cooperação entre membros de comunidades de informação do mesmo domínio que podem ter vindo de outras comunidades informacionais onde as conceitualizações são distintas. Mesmo que compartilhem da mesma linguagem natural, o vocabulário utilizado informalmente ainda pode conter elementos terminológicos que pertencem a comunidades informacionais específicas, não compartilhadas com outras comunidades. Na Engenharia de Software, são utilizadas modelagens conceituais para relacionar a estrutura com os detalhes de um problema de domínio e sua solução. Neste cenário, temos um exemplo do uso de um vocabulário comum e compartilhado para comunicação: os diagramas de relações de entidades e linguagens de modelagem orientada a objetos.

- Engenharia de Software: as ontologias para a descrição de informações

e sistemas podem contribuir na identificação de requisitos e inconsistências de um padrão escolhido. Também, nas especificações de componentes implementados no sistema (que podem ser utilizados para a manutenção, extensão e reutilização do software). Por exemplo, quando um componente já implementado corresponda com requisitos de uma outra tarefa especificada.

- Interoperabilidade: uma importante área para a aplicação de ontologia é a integração de sistemas de mesmo domínio e a capacidade de troca de informações em tempo real entre esses sistemas. Para que haja uma interoperabilidade de informações, todos os sistemas devem empregar o mesmo vocabulário ontológico como forma de padronização para o domínio de interesse.

- Recuperação da Informação: as técnicas de recuperação de informação dependem de uma codificação específica das informações ou da análise de textos. Porém, a codificação específica ou o vocabulário da busca do usuário podem não ser consistentes com o vocabulário da informação e possuir significados ambíguos. Com o uso de ontologias a recuperação da informação torna-se mais precisa, pois operam a busca em um nível semântico.

Segundo Guarino et al. (1998), as ontologias podem ser classificadas de acordo o objetivo de uso e com o contexto do domínio a ser aplicado. Os autores descrevem quatro principais tipos de ontologias de acordo com o apresentado na Figura 2.2:

- Ontologias de topo: são ontologias que descrevem conceitos muito gerais como espaço, tempo, matéria, objeto, evento, entre outros. Essas ontologias não dependem de um determinado problema ou domínio.

- Ontologias de domínio: têm a função de descrever o vocabulário relacionado a um domínio genérico, especializando os termos introduzidos na ontologia de topo. Por exemplo, ontologias descrevendo um domínio como medicina ou automobilismo.

- Ontologias de tarefa: estas descrevem uma tarefa ou atividade genérica dentro da ontologia de domínio, tais como prontuários médicos (no domínio de medicina) ou a compra de veículos (no domínio automobilístico).

- Ontologias de aplicação: são ontologias que descrevem uma aplicação dependente tanto de um domínio quanto de uma tarefa especifica. São muitas vezes especializações de ambas as ontologias relacionadas. Tais conceitos correspondem ao papel desempenhado por alguma entidade dentro de um domínio, durante a execução de uma determinada tarefa.

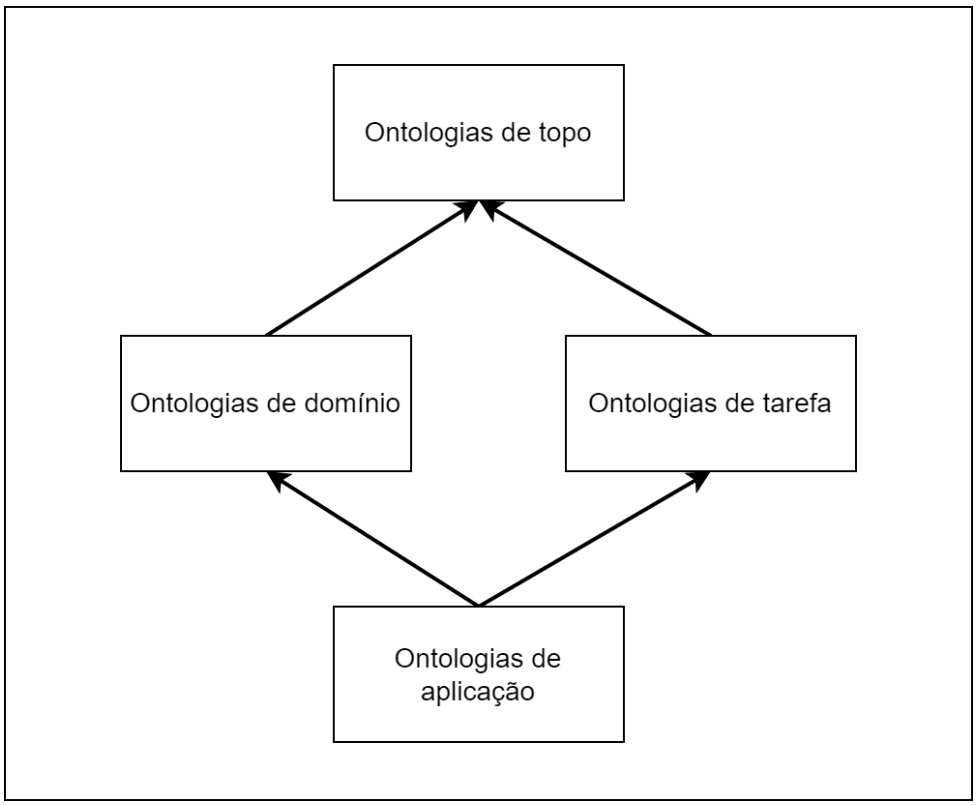


Figura 2.2: Classificação da ontologia. Adaptado de (Guarino et al., 1998).

2.2 Extensible Markup Language (XML)

A utilização da linguagem XML para a representação de dados estruturados está presente na proposta inicial da Web Semântica (Berners-Lee et al. (2001)). O XML é uma linguagem flexível e interoperável, permitindo a definição de regras para validação da sua estrutura. A regras são definidas em uma estrutura secundária denominada XML *Schema Definition* (XSD).

Com a utilização do XML e a sua estrutura correspondente para definição de vocabulários, a modelagem não especifica um nível de semântica esperado para o desenvolvimento da Web Semântica. Outras linguagens para a marcação de dados semânticos foram desenvolvidas apresentando uma maior expressividade em relação ao XML. Porém, a maioria das linguagens presentes na Web Semântica são codificadas utilizando a sintaxe e a representação do XML para a estruturação de dados em ontologias.

O W3C (2015) explica que não há uma divisão clara entre "vocabulários" e "ontologias" na Web Semântica. Os vocabulários definem os conceitos e relacionamentos usados para representar um domínio e podem ser utilizados para classificar e definir possíveis restrições ao uso dos relacionamentos. Os vocabulários podem ser muito complexos (com milhares de termos) ou muito simples (descrevendo apenas um ou dois conceitos).

A tendência é utilizar o termo "ontologia" para um conjunto de relaciona-

mentos mais complexo e formal, enquanto "vocabulário" é usado quando esse formalismo estrito não é necessariamente usado ou apenas em um sentido muito amplo.

2.3 Resource Description Framework (RDF)

O *Resource Description Framework* (RDF) visa solucionar o problema de legibilidade de dados da Web Sintática por máquinas e a criação de metadados que possibilitem descrever os dados contidos nas páginas da Web, permitindo o processamento automatizado de recursos da Web Semântica.

O RDF fornece um modelo formal de dados e sintaxe para codificar metadados para que possam ser processados por máquinas, com o objetivo de fornecer interoperabilidade entre aplicações que estão no mesmo domínio de conhecimento (Lassila et al., 1998).

Portanto, o RDF é apenas um modelo de metadados e utiliza o XML como um complemento para resolver problemas como codificação e armazenamento de arquivos. A sintaxe XML é apenas uma representação de várias alternativas possíveis para o modelo de metadados RDF.

Segundo Lassila et al. (1998) e Breitman (2000) o RDF pode ser utilizado em diversas áreas e aplicações:

- Descoberta de recursos para motores de pesquisas;
- Catalogação para a descrição de conteúdo e suas relações;
- Agentes de software inteligentes;
- Compartilhamento de conhecimento;
- Descrição de direitos de propriedade intelectual de páginas Web, preferências de privacidade de usuário e políticas de privacidade de um site;
- Informações sobre páginas Web (título, autor e data de criação);
- Conteúdo e classificação de figuras;
- Conteúdo para mecanismos de busca;
- Bibliotecas eletrônicas.

O modelo base do RDF é constituído por três princípios fundamentais: sujeito (recurso), predicado (propriedade) e objeto (valor). Os recursos são objetos ou coisas (livros, pessoas, lugares e etc). As propriedades podem ser consideradas atributos de recursos e também representam relacionamentos entre recursos. Por fim, os objetos são os valores correspondentes do relacionamento de um recurso ou propriedade, ou seja, são dados que representam o

conteúdo que está sendo descrito. Os objetos são os únicos elementos do RDF que podem ser um recurso ou um valor literal (Breitman, 2000).

Cada recurso e propriedade são identificados de maneira única e simples utilizando o *Universal Resource Identifier* (URI), que pode ser caracterizado por três aspectos: uniformidade, recurso e identificador (Isotani e Bittencourt, 2015). Uniformidade garante a utilização de recursos em contextos diferentes. O recurso é qualquer coisa que pode ser identificada por um URI. O identificador é a informação para identificar e diferenciar um recurso, podendo ser via nome, local ou qualquer característica de maneira única.

Além da utilização de URI Isotani e Bittencourt (2015) destacam que também podem ser utilizados para a identificação de recursos os padrões *Uniform Resource Locator* (URL), *Unified Resource name* (URN) e *International Resource Identifier* (IRI), sendo este último uma generalização do URI baseada nos caracteres ASCII.

A URI é essencial para garantir a identificação exclusiva para cada recurso e domínio. DuCharme (2013) destaca que a URI se assemelha muito a uma URL, podendo até haver uma página Web em seu endereço. A URI identifica recursos RDF como os identificadores exclusivos de uma tabela no modelo relacional, mas sua principal diferença é que são universalmente únicas, o que permite vincular dados de diferentes domínios por toda a Web.

O exemplo ilustrado na Figura 2.3 considera a representação de uma tripla da linguagem RDF no formato de grafo. O primeiro elemento em forma de arco representa o sujeito (recurso) identificado por um URI; o elemento de ligação representa o predicado (propriedade) também identificado por uma URI e por fim o objeto (valor) identificado pela forma de retângulo. Os valores relacionados ao sujeito e ao objeto são diferenciados por valores literais e não literais. Os valores literais podem conter uma *string*, um número ou um valor absoluto, porém os valores não literais representam os recursos ou propriedades.

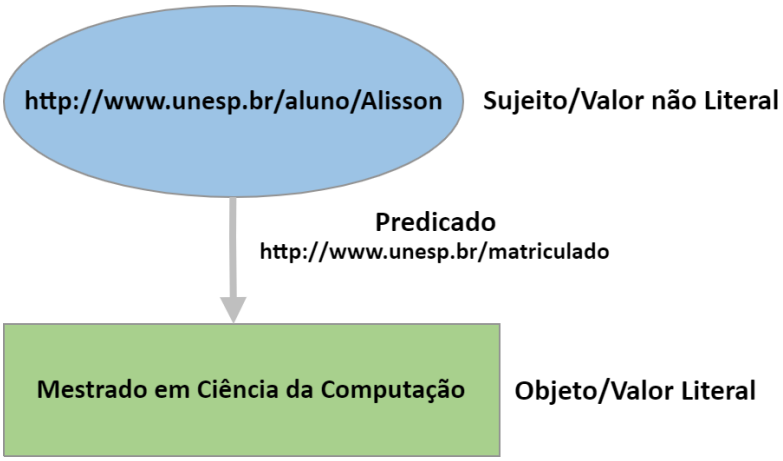


Figura 2.3: Representação gráfica de uma tripla RDF (Elaborado pelo autor).

Dessa maneira na Figura 2.3 temos a representação da tripla indicada pelo recurso http://www.unesp.br/aluno/Alisson, o predicado http://www.unesp.br/matriculado e o objeto com o valor "Mestrado em Ciência da Computação" sendo um valor literal.

Na Figura 2.4 podemos observar a representação da tripla utilizando o objeto como um valor não literal. Sendo assim, o objeto está assumindo a representação de um recurso formando um grafo dirigido e acíclico. Um documento RDF pode disponibilizar múltiplas triplas seguindo este formato.

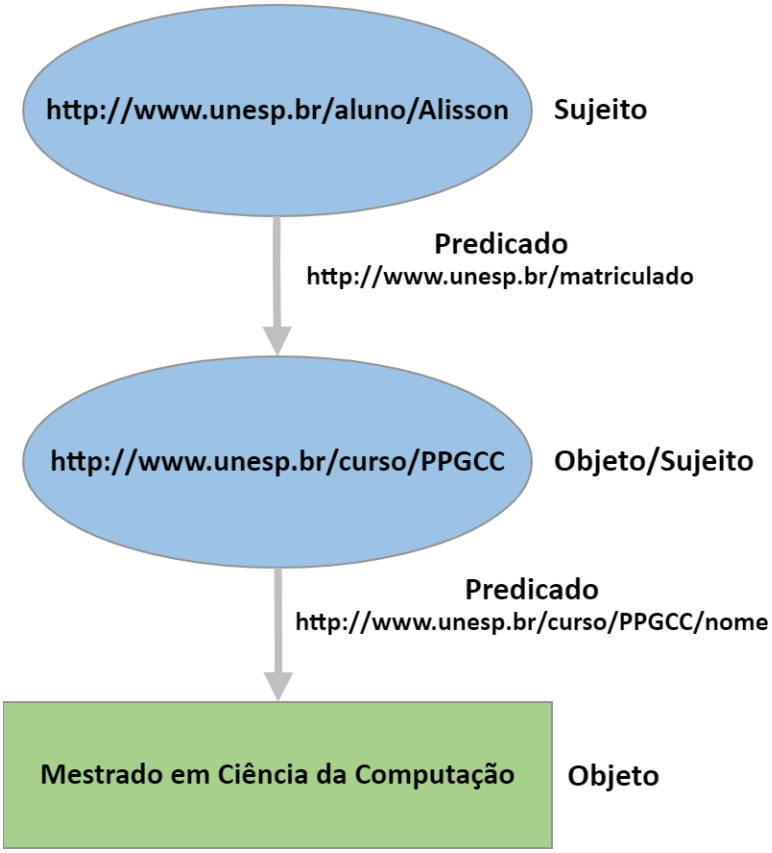


Figura 2.4: Representação gráfica tripla RDF com valor não literal (Elaborado pelo autor).

Nas Figuras 2.3 e 2.4 são ilustradas as estruturas conceituais do modelo de metadados do cenário utilizado como exemplo. Porém, para a criação e uso desses metadados descritos no modelo RDF é necessário utilizar uma sintaxe concreta para fins de criação e troca desses metadados.

A codificação do esquema RDF exige que o recurso da linguagem associe precisamente cada propriedade ao esquema que a define. Segundo Lassila et al. (1998) esse significado é fundamental para entender as instruções do modelo RDF e garantir que o processamento correto das informações e notações ocorra conforme o esperado. Inclui-se a isto a necessidade de serem interoperáveis, ou seja, é essencial que tanto o escritor quanto o leitor de uma declaração entendam o mesmo significado para os termos usados.

Na utilização da linguagem XML para codificação dos modelos RDF são utilizados os recursos de *namespace* da linguagem XML, sendo um esquema (dicionário) em que as definições e restrições de uso de propriedades são documentadas, ou seja, ao codificar o modelo RDF cada predicado utilizado na conexão dos elementos deve estar declarado no esquema ao ser utilizado. O código 2.1 representa o conteúdo apresentado na Figura 2.3, utilizando o XML para realizar a descrição.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<rdf:Description rdf:about="http://www.unesp.br/aluno/Alisson">
  <matriculado>Mestrado em Ciencia da Computacao</matriculado>
</rdf:Description>
```

Listing 2.1: Representação de uma tripla RDF utilizando a linguagem XML

2.4 RDF Schema (RDF-S)

O RDF representa os objetos e seus relacionamentos, além de possibilitar o fornecimento de uma representação semântica de sua estrutura utilizando a linguagem XML. Ocorre que, este modelo não oferece informações necessárias para as linguagens de ontologias.

O *RDF Schema* (RDF-S) é uma extensão semântica do RDF com o objetivo de fornecer um vocabulário mais expressivo para a modelagem de dados, provendo mecanismos para a descrição de grupos de recursos (classes) e suas relações utilizando o conceito de triplas (Brickley et al., 2014).

Segundo Breitman (2000) o RDF fornece um número limitado de elementos predefinidos, sendo necessário o desenvolvimento de vocabulários independentes para a expressividade de domínios particulares. O RDF-S fornece uma extensão para a ampliação do vocabulário, muito parecido com o conceito de classes no paradigma de programação orientado a objetos. Possibilitando que os recursos descritos no RDF-S sejam definidos como instâncias ou subclasses de classes.

Isotani e Bittencourt (2015) destacam que assim como o *XML Schema* é baseado no XML o mesmo ocorre com o RDF-S e RDF, sendo que o RDF-S também apresenta uma estrutura de triplas: sujeito (recurso), predicado (propriedade) e objeto (valor).

As classes no RDF-S são utilizadas para a descrição de recursos no documento RDF. Os membros de uma classe podem ser considerados com instâncias da classe e são identificados por IRIs podendo também ser descritos usando propriedades RDF (Brickley et al., 2014).

2.5 Ontology Web Language (OWL)

Segundo Isotani e Bittencourt (2015), existem basicamente duas maneiras de representar uma ontologia: a representação formal e a representação gráfica. A representação formal é utilizada para que as informações possam ser compreendidas por máquinas, enquanto a representação gráfica é utilizada para a melhor compreensão humana. Ambas as representações são essenciais para o uso de qualidade da ontologia.

Para a representação formal de ontologias as linguagens de codificação mais populares são o RDF e RDF-S, essas linguagens tem como objetivo a descrição de metadados e o relacionamento entre os dados proporcionando, uma representação semântica. Porém, tanto o RDF quanto o RDF-S possuem algumas limitações para a descrição de ontologias. Mesmo o RDF-S sendo uma extensão do RDF com o objetivo de enriquecer o vocabulário e fornecer mais elementos possibilitando realizar a descrição de ontologias, ainda assim esse modelo possui algumas limitações de expressividade, para apoiar o raciocínio computacional dos dados.

Identificado essa adversidade o W3C (2013) desenvolveu uma linguagem para a construção de ontologias denominada OWL (*Web Ontology Language*). Essa linguagem possui uma série de elementos que permitem a descrição de uma ontologia, satisfazendo o formalismo necessário da Web Semântica e tornando possível que máquinas possam compreender e responder à consultas de agentes inteligentes, usuários ou outros aplicativos através de descrições ontológicas.

De acordo com Segundo et al. (2017), a linguagem OWL possibilita a implementação computacional de todas as características necessárias da Web Semântica, além de representar uma uma série de propriedades que inserem lógica e axiomas nas relações existentes, tornando a descrição do contexto de um domínio com uma semântica formal muito representativa.

A linguagem OWL faz parte das tecnologias da Web Semântica que contemplam também o RDF, RDF-S e SPARQL e sua estrutura sintática principal é baseada em RDF/XML. Breitman (2000) sintetiza que a linguagem OWL foi projetada para atender as seguintes necessidades da Web Semântica:

- Construção de ontologias;
- Explicitar fatos de um determinado domínio; e
- Racionalizar sobre ontologias e fatos.

Isotani e Bittencourt (2015) destacam que: a OWL não é uma linguagem de programação e sim uma linguagem declarativa que descreve um determinado universo do discurso de forma lógica. A linguagem OWL também não é um

banco de dados, apesar dos documentos OWL armazenarem as instâncias do domínio descrito, a principal diferença entre um banco de dados está na semântica utilizada em cada um deles. Um banco de dados é considerado um mundo fechado, portanto ao inferir um determinado fato se este não existir no banco de dados é considerado como falso. Porém, em uma ontologia é considerado um mundo aberto, ou seja, se determinado fato não está presente na ontologia este é considerado como desconhecido.

De acordo com W3C (2012), as categorias sintáticas básicas para a modelagem de conhecimento usando OWL consistem em:

- Entidades: são elementos utilizados para modelar objetos do mundo real (classes, propriedades e indivíduos) e são identificados por IRIs. Esses elementos formam os termos primitivos de uma ontologia. Por exemplo: pode-se utilizar uma classe <*Pessoa*> para representar o grupo de todas as pessoas e uma propriedade "*parentOf*" pode ser utilizada para representar o relacionamento entre pai e filho, e por fim, podemos ter um indivíduo <*Alice*> utilizado para representar uma pessoa específica.

- Expressões: combinações de entidades que representam descrições complexas no domínio em que está sendo descrito, por meio de expressões criadas utilizando termos primitivos. Por exemplo: podemos representar um conjunto de todos os alunos e professores por meio da conjunção das classes <*Aluno*> e <*Professor*> criando a nova classe <*AlunoProfessor*>.

- Axiomas: são afirmações que podem ser utilizadas para fazer inferências sobre as entidades descritas no domínio da ontologia. Por exemplo: pode-se afirmar que uma instância da classe <*Aluno*> também é uma instância de <*Pessoa*>, sendo que <*Aluno*> <*SubClassOf*> <*Pessoa*>.

2.6 Protocol and RDF Query Language (SPARQL)

O processo de recuperação da informação é de suma importância em ambientes de sistema de informação. Esse processo consiste em identificar no conjunto de documentos de um sistema, quais atendem à necessidade de informação do usuário. O usuário de um sistema de recuperação de informação está interessado em recuperar a informação sobre um determinado assunto e não em recuperar dados que satisfazem sua expressão de busca.

Nos Sistemas Gerenciadores de Bancos de Dados (SGBD) que tem por objetivo a recuperação de todos os objetos ou itens que satisfazem precisamente às condições formuladas através de uma expressão de busca. Nesses sistemas o usuário precisa traduzir a sua necessidade em uma expressão de busca que tenta exprimir a semântica através de uma linguagem atribuída pelo sistema. Deste modo, toda a responsabilidade de busca semântica é transferida para o

usuário, que deve possuir um conhecimento prévio das expressões de busca do sistema (Ferneda, 2003).

O processo de recuperação da informação é um dos pilares da Web Semântica, pois, após a descrição semântica de um determinado domínio é preciso recuperar essa informação para que seja possível atender a necessidade do usuário e inferir novos conhecimentos de domínio. Para isso foi projetada a linguagem SPARQL (*Protocol and RDF Query Language*) que possibilita realizar consultas e buscas em dados estruturados em RDF e OWL.

O SPARQL contém recursos que possibilitam consultar padrões gráficos obrigatórios e opcionais, juntamente com suas conjunções e disjunções. Assim como as consultas podem ser expressas em estruturas gráficas RDF ou em estruturas nativas, os resultados das consultas SPARQL também podem retornar conjuntos de resultados nativos ou gráficos RDF. Quando utilizado o termo "Gráfico" ou "Grafo" RDF refere-se a estrutura de dados de triplas RDF e não a um gráfico visual.

O protocolo SPARQL fornece as seguintes especificações: uma linguagem para consulta de dados estruturados em RDF; um protocolo que estende a linguagem SPARQL para executar consultas distribuídas em diferentes terminais; uma especificação que define a semântica de consultas SPARQL estruturadas em RDF Schema, OWL ou RIF; um protocolo que define os meios para a transmissão de consultas arbitrárias e solicitações de atualização para um serviço de SPARQL; uma especificação que define um método de busca e descoberta; um vocabulário para descrever serviços SPARQL e um conjunto de testes (Segundo e Coneglian, 2016).

A linguagem de consulta SPARQL não é sobre um modelo, mas sim sobre os dados, sendo principalmente utilizada para a extração de dados em coleções públicas e privadas. A linguagem de consulta SPARQL não foi projetada para consultar dados relacionais, mas para consultar dados utilizando o padrão de modelo RDF, possibilitando também a integração de banco de dados de diferentes plataformas que armazenam dados de domínios em comum, e, facilitando a interoperabilidade de diferentes aplicações (DuCharme, 2013).

As consultas utilizando SPARQL são sintaticamente semelhantes as consultas SQL de bancos relacionais. Para a recuperação de dados utilizando SPARQL são necessárias basicamente três etapas: consulta; seleção e apresentação.

No código 2.2 é apresentado uma consulta SPARQL com o objetivo de retornar o nome e o e-mail de todos os alunos no domínio consultado.

```
PREFIX p: <http://www.unesp.br/aluno#>

SELECT ?aluno ?email
FROM <unesp.ttl>
WHERE { ?aluno p:email ?email . }
```

Listing 2.2: Representação de uma consulta SPARQL

Para iniciar uma consulta utilizando a linguagem SPARQL é necessário primeiramente conhecer a estrutura semântica de organização dos dados; o vocabulário do domínio e seus relacionamentos.

A primeira declaração feita na consulta SPARQL, porém não obrigatória é a utilização de um prefixo (*PREFIX*) que possibilita a utilização da URI do vocabulário que será utilizado para a recuperação da informação, sem a necessidade de referenciá-lo toda vez ao utilizar um predicado descrito no vocabulário. Deste modo, pode-se utilizar identificadores que abreviam o uso da URI.

O *SELECT* é o elemento relacionado à apresentação dos dados, neste fragmento são definidas as variáveis retornadas na consulta.

A cláusula *FROM* é utilizada para definir em qual fonte de dados é realizada a consulta, mas quando utilizado um terminal para consultas em um domínio específico o uso do *FROM* não se torna obrigatório, pois já é definido de forma implícita na consulta.

Por fim, o comando *WHERE* permite realizar o filtro da seleção obtida na fonte de dados de acordo com as relações definidas na consulta.

Na Figura 2.5 é exemplificado de forma gráfica o funcionamento de uma consulta SPARQL: primeiro é realizada a seleção na fonte de dados; depois é realizado um filtro de acordo com as regras especificadas e, por fim, selecionados os dados extraídos que são retornados na consulta.

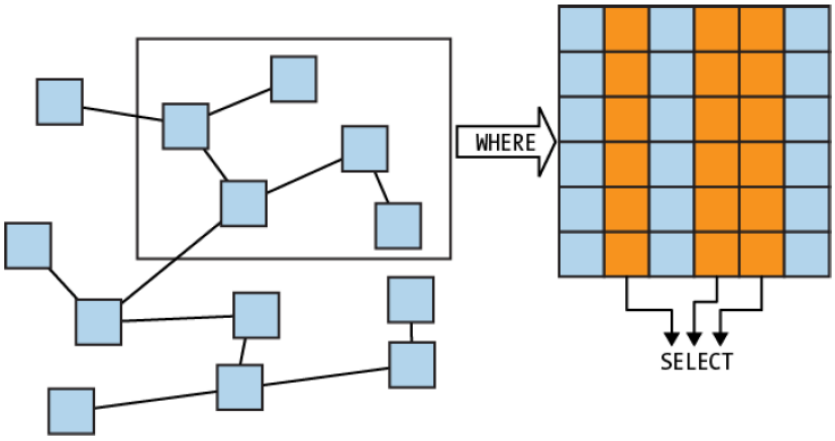


Figura 2.5: Funcionamento das cláusulas *SELECT* e *WHERE* em uma consulta SPARQL (DuCharme, 2013).

Com a utilização linguagem SPARQL também é possível realizar inferências, criar novas relações, realizar o agrupamento de dados, operações matemáticas, ordenação, entre outras opções, por meio de diferentes operadores disponíveis na linguagem SPARQL (DuCharme, 2013).

Além da cláusula *SELECT* existem mais três formas de realizar consultas em SPARQL:

- CONSTRUCT: tem como finalidade retornar triplas RDF. É utilizada para a extração de valor com o objetivo de criar novas triplas RDF, converter e copiar.
- ASK: o processador de consultas verifica se um determinado padrão de grafos descreve um conjunto de triplas RDF de uma fonte de dados, retornando valores booleanos.
- DESCRIBE: identifica e descreve as triplas RDF de um recurso específico. O processador de consultas decide quais triplas deve retornar e descrever na consulta.

2.7 Uso de Ontologias na Engenharia de Software

Gasevic et al. (2009) destacam que sendo o software um repositório de conhecimento histórico que está diretamente relacionado ao domínio do aplicativo, é necessário que seja possível compartilhar o conhecimento armazenado em todos os diferentes processos que envolvem a Engenharia de Software. Para que esse compartilhamento seja alcançado, é necessária uma definição explícita desse conhecimento para que as máquinas também possam interpretar e inferir os conhecimentos gerados ao longo dos processos.

A comunidade de Engenharia de Software reconheceu que a ontologia é uma tecnologia promissora para a abordagem de problemas relacionados aos processos de Engenharia de Software. Deste modo, as ontologias podem desempenhar papéis fundamentais nos processos de modularização, distribuição, reutilização de componentes e integração de sistemas de software de mesmo domínio.

As metodologias de desenvolvimento de software, como o MDD (Model Driven Design), favorecem a modelagem de domínio centrada no conhecimento, do qual diferentes implementações podem ser derivadas. Essa modelagem favorece a interoperabilidade de aplicações que possuem estruturas e terminologias próprias para a descrição de um mesmo domínio (Hesse, 2005).

A organização OMG (Object Management Group), com o objetivo de definir padrões para o uso de ontologias na Engenharia de Software, definiu o padrão

ODM (Ontology Definition MetaModel), que é uma especificação para a representação de conhecimento semântico nos processos de Engenharia de Software, possibilitando integrar linguagens de ontologia no processo de desenvolvimento de software. O padrão ODM pode ser aplicável na representação do conhecimento, modelagem conceitual, desenvolvimento formal de taxonomia e definição de ontologia (OMG, 2014).

Happel e Seedorf (2006) destacam que o conhecimento do domínio e o uso de ontologias podem ser utilizados em todo o ciclo de vida da Engenharia de Software. As abordagens proeminentes são descritas a seguir.

- **Engenharia de requisitos**: As ontologias podem ser utilizadas para descrever formalmente os requisitos de conhecimento do domínio. A ontologia pode representar um grau de expressividade semântico e estruturado, possibilitando uma abordagem evolutiva para requisitos de domínio. O processo de validação e verificação pode reutilizar o conhecimento representado em uma ontologia na fase de desenvolvimento e implantação.

- **Reutilização de componentes**: No processo de reutilização de componentes as ontologias podem auxiliar na descrição das funcionalidades de maneira semântica em suas propriedades, permitindo consultas sobre componentes.

- **Manutenção de software**: o uso de ontologias fornece uma camada de integração e interoperabilidade de dados de diversas fontes, unificados em um modelo semântico. Além de proporcionar uma expressividade semântica dos dados é possível ainda inferir novos conhecimentos que não foram explicitamente descritos no relatório de defeito reportado.

Com relação ao processo de manutenção de software Gasevic et al. (2009) também enfatizam que o uso de ontologias podem proporcionar diversos benefícios. Ao desenvolver um software são necessários conhecimentos sobre o domínio da aplicação, tecnologias, procedimentos de testes e a compreensão dos requisitos. Porém, todo esse conhecimento não é registrado e documentado de maneira formal, dificultando compreender integralmente o sistema que está sendo mantido. Deste modo, as ontologias podem contribuir para a inferência de novos conhecimentos e para a recuperação da informação de maneira mais eficiente, facilitando a avaliação do conhecimento relevante para o contexto da aplicação.

O uso de ontologias em Engenharia de Software possui um grande potencial para o progresso dos processos de construção do software. Happel e Seedorf (2006) em sua pesquisa categorizam o uso das ontologias na Engenharia de Software. Os autores distinguem o uso das ontologias em dois cenários.

O primeiro cenário remete ao uso de ontologias em tempo de execução e em tempo de desenvolvimento. No segundo cenário é analisado o problema do domínio que o software busca resolver, classificando em aspectos de infraestrutura e de software.

Na Figura 2.6 é apresentado a classificação do uso das ontologias na Engenharia de Software.

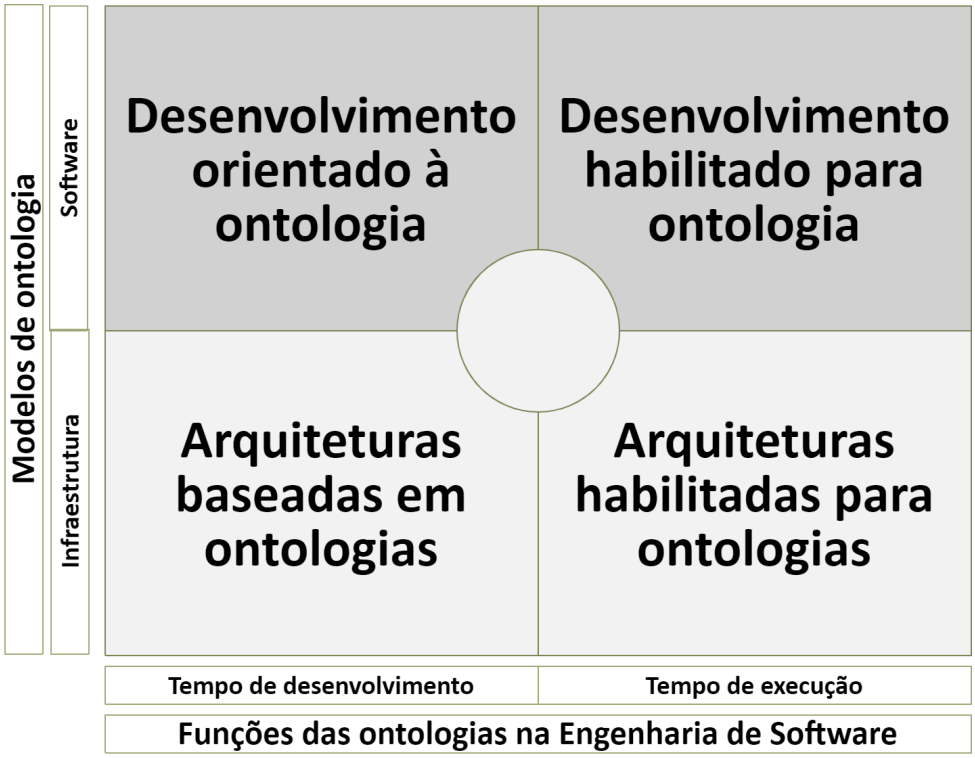


Figura 2.6: Classificação do uso de ontologias na Engenharia de Software. Adaptado de (Happel e Seedorf, 2006).

- **Desenvolvimento orientado à ontologia** (*Ontology-driven development (ODD)*): emprega o uso de ontologias em tempo de desenvolvimento para descrever o problema de domínio. O uso de linguagens ontológicas como o RDF e OWL permite reduzir a ambiguidade do idioma, possibilitando o processo de validação e verificação automatizada. Deste modo, as ferramentas baseadas na linguagem UML podem ser estendidas para oferecer suporte à criação de vocabulário de domínio com ontologias.

- **Desenvolvimento habilitado para ontologia** (*Ontology-enabled development (OED)*): essa categoria também é utilizada no momento do desenvolvimento, mas visa apoiar as tarefas realizadas pelos desenvolvedores de software, tais como: pesquisa de componentes, reutilização de componentes e manutenção de software.

- **Arquiteturas baseadas em ontologias** (*Ontology-based architectures (OBA)*): as arquiteturas neste cenário utilizam as ontologias como um artefato

primário, constituindo a lógica central da aplicação e as regras de negócio.

- **Arquiteturas habilitadas para ontologia** (*Ontology-enabled architectures (OEA)*): as ontologias são aplicadas para fornecer suporte à infraestrutura de um sistema de software adicionando uma camada semântica sobre a camada sintática existente. Deste modo, possibilitam a recuperação da informação semântica e a descrição de serviços baseados em um vocabulário semântico formal.

2.8 Considerações Finais

Neste capítulo foram estudados os principais conceitos e tecnologias para a utilização de ontologias. Foram destacadas três tecnologias que são base para projetos que visam a estruturação e a recuperação semântica dos dados para um determinado domínio, ganhando maior destaque o XML, RDF, RDFS, ontologias e as consultas SPARQL.

Com o uso de ontologias é possível descrever formalmente os dados e metadados de um domínio, possibilitando a interpretação dos dados de um domínio não apenas por humanos, mas também por máquinas e agentes inteligentes. Com a utilização de consultas SPARQL em uma ontologia é possível inferir novos conhecimentos e realizar a recuperação da informação de maneira semântica.

Podemos identificar a sinergia presente entre a modelagem de conhecimento utilizando ontologias para a abordagem de problemas relacionados a Engenharia de Software. As ontologias podem ser aplicáveis na representação de conhecimento, modelagens conceituais e na interoperabilidade de conhecimentos entre aplicações heterogêneas. Os processos de engenharia de requisitos, reutilização de componentes e a manutenção de software também podem ser beneficiados com o uso de ontologias.

A concepção e o uso de ontologias é a principal tecnologia para a criação de aplicativos que suportem e manipulem dados semânticos. As ontologias contribuem na solução de problemas em aplicações tecnológicas que utilizam base de dados como representação formal do conhecimento.

Na representação formal do conhecimento sem a utilização de ontologias pode-se destacar algumas dificuldades, tais como: nas bases de dados as premissas básicas estão implícitas impedindo a reutilização e o compartilhamento do conhecimento representado; não existe um modelo genérico com o qual podemos construir banco de dados reutilizando um modelo de conhecimento existente; e por fim, não há uma tecnologia que permita estender de forma rápida uma base de dados, ou seja, unir conhecimentos de domínios complementares ao domínio da aplicação (Isotani e Bittencourt, 2015).

CAPÍTULO 3

Rastreamento de Defeitos

Neste capítulo são apresentadas ferramentas da Engenharia de Software utilizadas para o rastreamento de defeitos. São discutidas as diferentes definições de defeitos e exploradas duas ferramentas que contribuem para o rastreamento destes. Por último, são realizadas as considerações finais deste capítulo.

3.1 Definição de Defeitos

Um erro, defeito, falha ou engano no software é denominado *bug* e comumente indica um comportamento inesperado de um sistema de software. Porém, existem definições diferentes para cada um desses termos na Engenharia de Software. O IEEE (1983) distingue os termos como:

- **Engano (*mistake*)**: É uma ação humana que produz um resultado incorreto, ou seja, uma ação incorreta feita pelo programador, podendo introduzir um defeito e consequentemente produzir uma falha na aplicação.
- **Defeito (*fault*)**: O defeito é considerado uma etapa, processo ou modelagem de dados incorreta. Por exemplo, uma instrução ou comando incorreto no programa, ou um curto-circuito ou um fio rompido em um dispositivo ou componente de hardware.
- **Falha (*failure*)**: A falha é a incapacidade de um sistema ou componente de software executar as funções especificadas de acordo com os requisitos. Delamaro et al. (2013) concluem que uma falha é a produção de uma saída incorreta com relação à especificação.

- **Erro (*error*)**: Ocorre quando um valor ou condição esperada divergem do especificado ou da definição teórica correta, ou seja, qualquer resultado que não seja o esperado na execução de um programa. No software, um erro é um estado inconsistente da máquina.

Na Figura 3.1 são representadas as diferentes definições de *bug*, sendo que, a partir do engano do desenvolver inicia-se a introdução de um defeito, posteriormente produzindo uma falha e manifestando ou não um erro na aplicação.

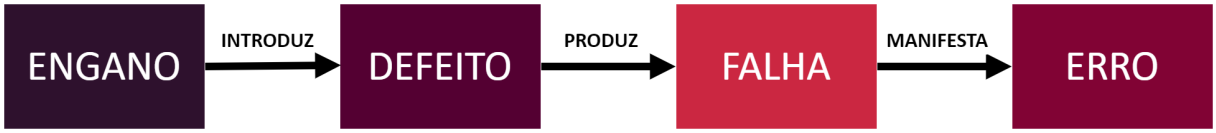


Figura 3.1: Engano x Defeito x Falha x Erro (Elaborado pelo autor).

3.2 Sistemas de Rastreamento de Defeitos

Na fase de evolução e manutenção do software, um importante artefato de geração de conhecimento no processo para o rastreamento e controle de inconsistências são os relatórios de defeitos (*Bugs Report*). Esses documentos têm como objetivo descrever as ocorrências em que o software não se comporta como o esperado. Zibran (2016) destaca que os relatórios de defeitos ajudam no processo de triagem, sendo que esses relatórios possuem informações valiosas para Engenharia de Software.

Os softwares denominados *Bug Tracking System* (BTS), fornecem um sistema para rastreamento e armazenamento de relatórios de defeitos, possibilitando que sejam reportadas as inconsistências encontradas durante todo o ciclo de vida do software.

Os sistemas BTS fornecem uma plataforma completa para registrar, rastrear, pesquisar, comentar, descrever e classificar defeitos. Segundo Uddin et al. (2017) um repositório de defeitos é considerado uma base de dados indispensável no processo de Engenharia de Software e permite que usuários e desenvolvedores possam relatar e acompanhar as inconsistências ocorridas no projeto de software, contribuindo para garantia da qualidade do projeto.

Algumas plataformas de hospedagem e controle de versão de código-fonte também implementam funcionalidades para rastreamento e armazenamento de defeitos, mesmo originalmente não sendo sua responsabilidade principal.

Para iniciar o processo de registro de um defeito em um sistema de rastreamento de defeitos ou mesmo em uma ferramenta que possua tal funcionalidade, o usuário preenche um formulário para fornecer detalhes sobre o

ocorrido. Geralmente, este formulário possui vários atributos para que possa ser detalhado o defeito com maior precisão, contribuindo para a identificação dos artefatos de código-fonte pela equipe de desenvolvimento. Após o relato no sistema BTS, a equipe de desenvolvimento inicia a resolução dos defeitos de acordo com sua classificação, sendo que geralmente a quantidade de defeitos registrados é demasiadamente alta (Gujral et al., 2015).

Os relatórios de defeitos de software contêm diversos atributos que contribuem para a compreensão do problema ocorrido. Os atributos expressam características qualitativas e quantitativas, os quais são preenchidos durante o relato do defeito ou durante o processo de correção pela equipe de desenvolvimento. Nas próximas subseções são detalhados os ciclos de vida de um relato de defeito nas plataformas Bugzilla e GitHub.

3.2.1 Bugzilla

Na Figura 3.2 é apresentada a visão geral do conteúdo de um relatório de defeito descrito no Bugzilla.

Figura 3.2: Relatório de defeito do sistema Bugzilla (Eclipse, 2020).

Um relatório de defeito no sistema Bugzilla possui os seguintes atributos:

- **Sumário (*Summary*)**: Uma descrição sucinta do defeito, exibido no ca-

beçalho à direita do identificador do defeito.

- **Estado (*Status*)**: Indica qual o *status* atual de um defeito de acordo com o seu progresso. Os status definem os estágios do defeito, desde a sua abertura (Não Confirmado (*Unconfirmed*), Confirmado (*Confirmed*) e Em Progresso (*In Progress*)) até a sua finalização (Resolvido (*Resolved*) e Verificado (*Verified*)).

- **Apelido (*Alias*)**: Um nome curto para descrever um defeito, geralmente utilizado para ser usado no lugar do identificador do defeito.

- **Produto e componente (*Product and Component*)**: Os defeitos são segmentados em produtos e componentes, podendo um produto ter um ou mais componentes vinculados.

- **Versão (*Version*)**: O campo referente à versão contém o número de versão ou nome referente ao produto, normalmente utilizado para indicar as versões afetadas pelo defeito reportado.

- **Hardware (Plataforma e SO)**: Este atributo indica em qual ambiente computacional o defeito foi encontrado, podendo ser informado a plataforma e o sistema operacional.

- **Importância (prioridade e gravidade) (*Importance (Priority and Severity)*)**: Refere-se à importância de um defeito relatado sendo composto pela combinação de sua prioridade e gravidade. O atributo prioridade é utilizado pelos gerentes das equipes de desenvolvimento para definir a ordem de priorização dos defeitos, os valores padrões consideram os níveis de prioridade de 1 à 5 (P1 a P5). O atributo que descreve a gravidade do problema reportado pode ser considerado como *blocker* (defeito que torna a aplicação inutilizável) ou trivial (defeito que não influencia no uso da aplicação), podendo também indicar quando é feita uma solicitação de melhoria no software.

- **Marco Alvo (*Target Milestone*)**: Indica a versão futura em que o defeito será resolvido. Este atributo pode ser utilizado para incluir as versões futuras com qualquer sequência de texto que indica um marco no projeto.

- **Atribuição (*Assigned To*)**: O usuário responsável pela correção do defeito reportado.

- **Contato de controle de qualidade (*QA Contact*)**: Contato do usuário responsável pela garantia de qualidade do produto.

- **URL**: Refere-se ao endereço relacionado ao erro, não sendo obrigatório o preenchimento desse campo.

- **Quadro branco (*Whiteboard*)**: Um atributo disponível para adicionar notas e rótulos relacionados ao defeito.
- **Palavras-chave (*Keywords*)**: As palavras-chave são definidas para a marcação e classificação de defeitos.
- **Tags pessoais (*Personal Tags*)**: É um atributo semelhante as palavras-chave, porém são de uso particular do usuário, sendo visíveis apenas aos criadores da tag.
- **Dependências (dependências e bloqueios) (*Dependencies (Depends On and Blocks)*)**: Informa quais outros defeitos possuem dependência direta com o defeito em foco, ou seja, o defeito não pode ser corrigido a menos que os defeitos que possuem dependência sejam resolvidos.
- **Relatado (*Reported*)**: Usuário e data de registro da ocorrência do defeito.
- **Modificado (*Modified*)**: Informação da data e hora em que houve a alteração do registro da ocorrência do defeito.
- **Lista CC (*CC List*)**: Lista de usuários que recebem notificações por e-mail quando o defeito é modificado.
- **Bandeiras (*Flags*)**: Sinalizador utilizado para definir o estado de defeitos ou anexos.
- **Rastreamento de tempo (*Time Tracking*)**: Atributo utilizado para acompanhar o tempo do ciclo de vida de um defeito, desde a sua ocorrência no BTS até a sua resolução.
- **Anexos (*Attachments*)**: Podem ser anexados arquivos relacionados aos defeitos. Por exemplo, casos de testes, imagens da ocorrência e outros arquivos que possam contribuir para a compreensão do defeito ocorrido.
- **Comentários adicionais (*Additional Comments*)**: São comentários que podem ser incluídos pelos usuários durante o ciclo de vida do defeito, com o objetivo de contribuir para a sua resolução.

O ciclo de vida de um defeito ou também conhecido como fluxo de trabalho, considera diferentes estados nos quais um relatório de defeito pode pertencer. Na Figura 3.3 é ilustrado o ciclo de vida de um defeito no sistema Bugzilla.

O estado do defeito pode ser classificado em dois estados base: defeitos abertos e defeitos fechados. Os defeitos abertos são ocorrências que ainda não foram resolvidas, este estado pode possuir três diferentes transições.

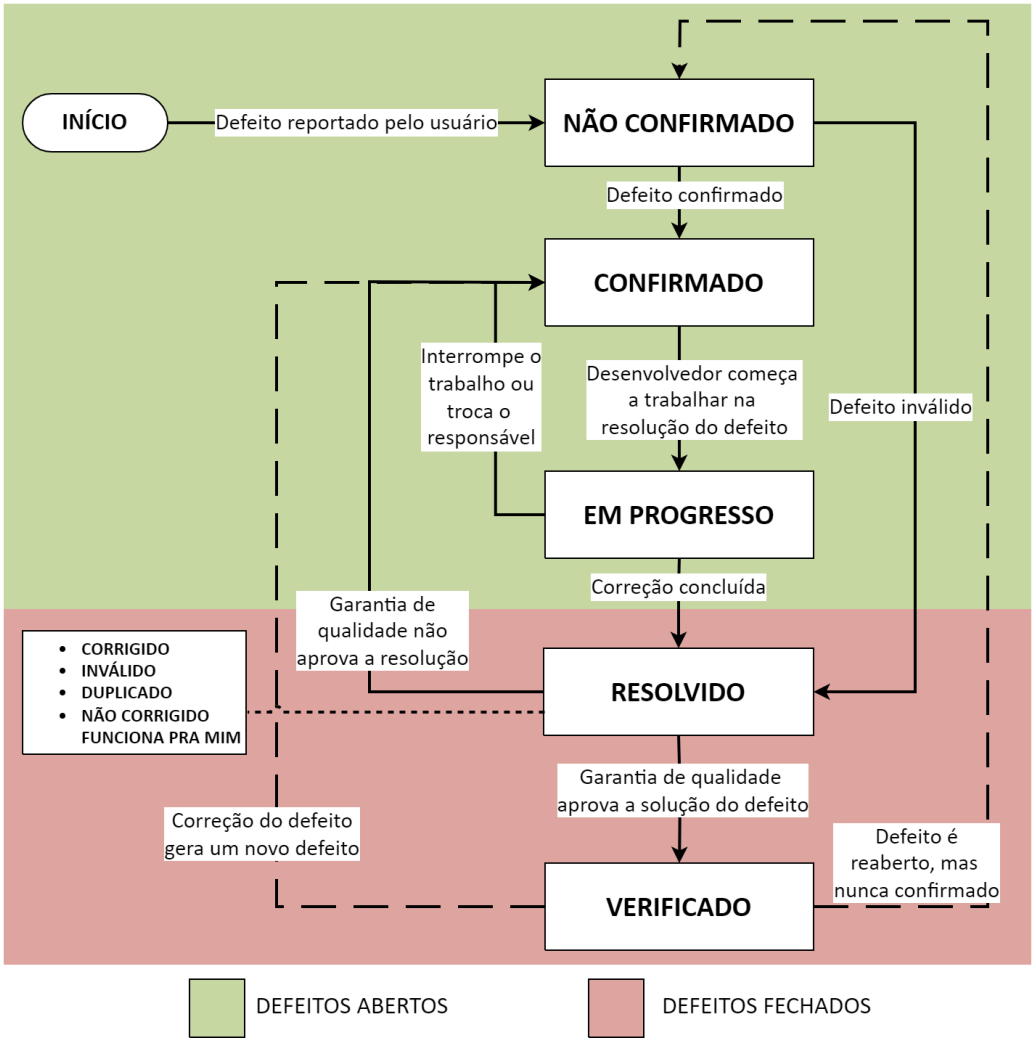


Figura 3.3: Ciclo de vida de um defeito adaptado do sistema Bugzilla (1998).

Os defeitos abertos são rotulados como “não confirmado”, quando o defeito foi relatado recentemente e necessita de um usuário com permissões exclusivas para verificar se a ocorrência é válida, podendo alterar o seu estado para “confirmado” ou “resolvido”. Os defeitos no estado “confirmado” são atualizados para “em progresso”, quando algum desenvolvedor começa a trabalhar para a resolução do defeito reportado. O defeito com estado “em progresso” pode retornar novamente para o estado “confirmado” quando houver a troca do usuário responsável por resolver o defeito, ou, atualizado para “resolvido” quando a correção estiver concluída.

O segundo estado base nomeado como “defeitos” fechados possui outros dois estados transacionais, “resolvido” e “fechado”. O estado “resolvido” ocorre quando foi encontrada uma solução para o defeito reportado e neste momento aguarda a verificação do controle de qualidade. Neste estado os defeitos reportados podem ser reabertos alterando o estado para “confirmado”, caso a correção do defeito não seja aprovada pelo controle de qualidade. Ainda no estado “resolvido” o defeito pode receber algumas marcações que categorizam

a sua solução:

- **Corrigido**: O defeito foi corrigido, verificado e testado.
- **Inválido**: A ocorrência reportada não é considerada um defeito.
- **Não corrigido**: O defeito reportado nunca será corrigido.
- **Duplicado**: O defeito foi reportado de maneira duplicada, ou seja, o defeito reportado já existe no repositório.
- **Funciona para mim**: Todas as tentativas de reproduzir o defeito para que seja feita a sua correção foram fracassadas e a leitura do código-fonte do componente vinculado ao defeito reportado não expõe o comportamento descrito no relato.

Caso a correção do defeito reportado seja "aprovada" o estado do defeito é transferido para o seu estado final "verificado". Neste estado o controle de qualidade que analisou o defeito e sua correção, aprova que essa seja a resolução apropriada para a ocorrência relatada. Deste modo, o ciclo de vida do defeito chega ao fim.

3.2.2 GitHub

O GitHub é uma plataforma para hospedagem e controle de versão de artefatos de código-fonte mundialmente utilizada. O GitHub possui uma funcionalidade para criação de *issues* (problemas) que são utilizadas para rastrear tarefas, comentários, ideias e defeitos encontrados nos artefatos de código-fonte do projeto (GitHub, 2022a).

Para iniciar o processo de registro de um defeito na plataforma GitHub o usuário inicia preenchendo um formulário para fornecer detalhes sobre o defeito ocorrido. O GitHub, difere dos sistemas BTS principalmente por não possuir um formulário rico e detalhado de atributos, pois essa não é a principal funcionalidade da plataforma.

As descrições de defeitos no GitHub podem conter imagens, trechos de código-fonte, *stack trace* ou qualquer outro elemento de acordo com a linguagem de marcação *Markdown*. O texto na plataforma GitHub é livre e não estruturado, dificultando a identificação do defeito. Ao relatar um defeito o usuário precisa incluir um rótulo que identifica que este relato é um defeito. Na Figura 3.4 é apresentada a visão geral do conteúdo de um defeito descrito no GitHub para o projeto AutoMapper (AutoMapper, 2022a).

v11 InvalidOperationException: Stack Empty on mapping which used to generate a subquery #3869

New issue

Closed **Erythnul** opened this issue on 26 Jan · 12 comments · Fixed by #3870

Erythnul commented on 26 Jan

Version: 11.0.0

Gist

https://gist.github.com/Erythnul/38a29c9012a3fdcc77fe117c51386ce9

Summary

Hi!

I've been reworking some of our more interesting mappers to start our upgrade to 11.0.0 and have run into the problem outlined in the gist. I can open a PR with a failing unit or integration test if necessary.

Looking forward to hearing from you!

Kind regards.

Expected behavior

In 10.1.1 this would generate an expression like

```
EnumerableQuery<AEntity>.Select(
    dtoAEntity => new Object_1451902296___DtoSubWrapper#DtoSub_Id
    {
        __DtoSubWrapper#DtoSub = dtoAEntity.BEntity.CEntities.FirstOrDefault(x => x.Id == dtoAEntity.CEnt:
        Id = dtoAEntity.Id
    }).Select(
    dtoLet => new Dto
    {
        DtoSubWrapper = new DtoSubWrapper
```

Assignees

No one assigned

Labels

Bug

Projects

None yet

Milestone

11.0.1

Development

Successfully merging a pull request may close this issue.

Handle identity lambda resolvers with ProjectTo sub...
AutoMapper/AutoMapper

4 participants

Figura 3.4: Relatório de defeito na plataforma GitHub (AutoMapper, 2022b).

Um relatório de defeito na plataforma GitHub possui os seguintes atributos:

- **Título (*Title*)**: Uma descrição sucinta do defeito, exibido no cabeçalho à esquerda da identificação do defeito.
- **Sumário (*Summary*)**: Um texto livre com a descrição do defeito, podendo conter imagens, trechos de código-fonte, *tack trace* ou qualquer outro elemento.
- **Estado (*Status*)**: Indica qual o status atual de um defeito de acordo com o seu progresso. Os status definem os estágios do defeito podendo ser Aberto (*Open*) ou Fechado (*Closed*).
- **Responsáveis (*Assignees*)**: O usuário responsável pela correção do defeito reportado.
- **Etiquetas (*Labels*)**: Rótulo utilizado para definir a classificação do defeito. Para a classificação de defeitos no GitHub a plataforma possui alguns rótulos padrões (GitHub, 2022b):
  - ***bug***: Indica um defeito inesperado ou comportamento involuntário.
  - ***documentation***: Indica a necessidade de aprimoramentos ou adições à documentação.
  - ***duplicate***: Indica defeitos, *pull requests* ou discussões.
  - ***enhancement***: Indica novas solicitações de recurso.
  - ***good first issue***: Indica um bom defeito para contribuidores principiantes.
  - ***help wanted***: Indica que um mantenedor deseja ajudar em um defeito ou uma *pull request*.
  - ***invalid***: Indica que um defeito, *pull request* ou discussão já não é relevante.
  - ***question***: Indica que um defeito, *pull request* ou discussão precisa de mais informações.
  - ***wontfix***: Indica que o trabalho não continuará em um defeito, *pull request* ou discussão.
- **Projetos (*Projects*)**: Um atributo disponível para relacionar o defeito ao quadro de um projeto.
- **Marco Alvo (*Milestone*)**: Indica a versão futura em que o defeito será resolvido. Este atributo pode ser utilizado para incluir as versões futuras indicando um marco no projeto.

- **Comentários (*Comments*)**: São comentários que podem ser incluídos pelos usuários durante o ciclo de vida do defeito, com o objetivo de contribuir para a sua resolução.

O ciclo de vida de um defeito na plataforma GitHub pode ser diferente de acordo com o projeto. Os usuários não ficam restritos a um único fluxo de trabalho definido pela plataforma e podem trabalhar de acordo com a metodologia que julgarem mais adequada. O ciclo de vida de um defeito no sistema Bugzilla apresentado na Figura 3.3 também pode ser implementado nos projetos do GitHub.

O repositório *Botframework Solutions* (Microsoft, 2022b) disponibiliza uma descrição do fluxo de trabalho ao reportar um novo defeito no projeto. A Figura 3.5 apresenta o fluxo utilizado. Apesar das particularidades de cada plataforma, podemos observar semelhanças no ciclo de vida de um defeito utilizado no Bugzilla.

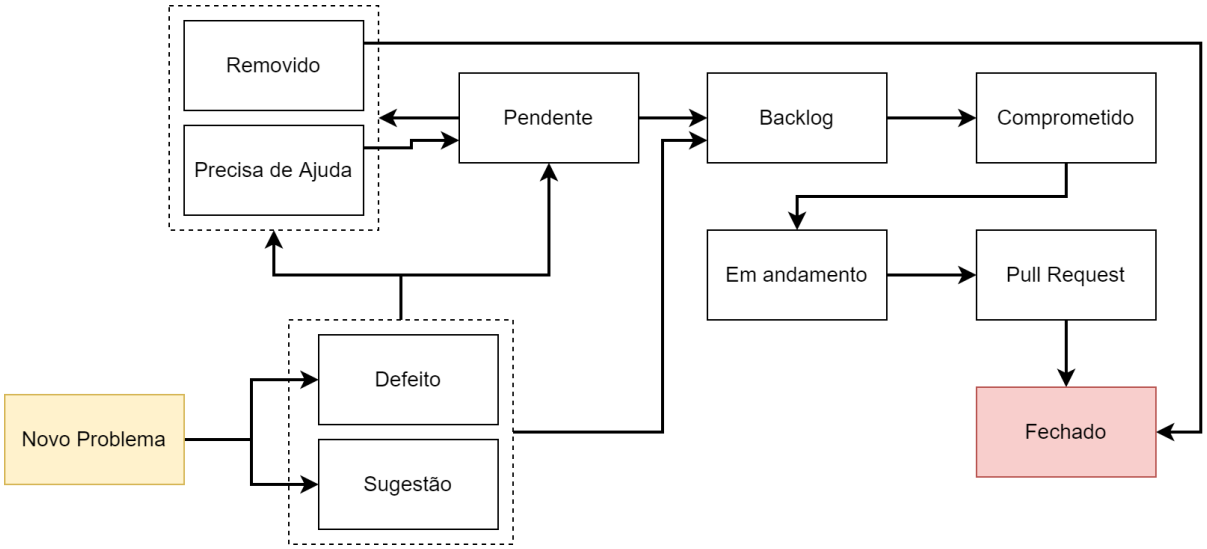


Figura 3.5: Ciclo de vida de um defeito do projeto. Adaptado de (Microsoft, 2022b).

O fluxo apresentado na Figura 3.5 possui sete etapas:

1. Ao reportar um novo defeito no projeto o usuário escolhe entre dois rótulos para classificação: “Defeito” ou “Sugestão”.

2. O defeito é triado e classificado de acordo com os seguintes critérios:

   - O defeito pode receber rótulo “Removido” caso a ocorrência não seja considerada um defeito ou sugestão.
   - O defeito pode receber o rótulo “Precisa de Ajuda” caso a ocorrência seja uma questão relevante da comunidade.

- O defeito pode receber o rótulo “Pendente” quando necessita de mais informações para o entendimento do relato.

3. Ao ser aprovado o defeito recebe o rótulo *“Backlog”* para ser revisado e analisado no próximo planejamento.

4. O rótulo “Comprometido” é adicionado quando o defeito está agendado para a próxima versão.

5. Se o defeito já estiver em andamento, o rótulo “Em Andamento” é aplicado.

6. Quando o defeito estiver corrigido uma nova solicitação *pull request* é aberta para realizar a atualização das modificações no projeto.

7. Ao finalizar o processo o defeito recebe o status de “Fechado”.

3.3 Considerações Finais

Neste capítulo foi conceitualizada a definição de defeito. O entendimento do conceito de defeito é importante para que seja possível compreender e identificar quais dados podem ser registrados e coletados durante o processo de um registro de defeito em um BTS. Foram detalhados os atributos presentes em um relatório de defeito do sistema Bugzilla e GitHub, além do ciclo de vida em cada ferramenta.

CAPÍTULO 4

Trabalhos Relacionados

Considerando que este trabalho propõe um modelo para localização de defeitos apoiado por ontologia e reconhecimento de entidades, foram estudados trabalhos que contribuam para o desenvolvimento do modelo proposto no capítulo 5. Neste capítulo, são apresentados trabalhos relacionados ao processo de localização de defeitos e o uso de ontologia para a modelagem do código-fonte. Posteriormente, são analisados os trabalhos relacionados e as lacunas encontradas.

De acordo com a pesquisa realizada sobre o processo de localização de defeitos e ontologias, foram selecionados os seguintes trabalhos relacionados pertinentes com a proposta de projeto apresentada. Os trabalhos abaixo selecionados estão categorizados de acordo com a área do projeto.

4.1 Manutenção de Software com o Apoio de Ontologias

- *Empowering software maintainers with semantic web technologies* (Witte et al., 2007)

  Este trabalho elenca alguns dos problemas relacionados ao processo de manutenção de software. Um dos principais desafios dos mantenedores de software é a necessidade de compreender os artefatos relacionados a este processo, que geralmente, estão desconectados. Os artefatos incluem: arquivos de código-fonte e documentos de projetos.

  Estabelecer uma conexão semântica entre esses artefatos é essencial para a otimização da tarefa de manutenção, além da representação do conhecimento em estruturas formais com anotações semânticas. Três casos relativos ao processo de manutenção de software são considerados

nessa abordagem.

- Identificação de falhas de segurança no código-fonte.
- Estabelecer uma conexão entre os artefatos, integrando informações de documentações e código-fonte.
- Compreensão da estrutura arquitetural do projeto, identificando os principais componentes e propriedades.

Para a aplicação da abordagem são utilizadas tecnologias da Web Semântica para fornecer uma representação unificada para a exploração, consulta e conhecimento dos artefatos relacionados. Neste trabalho, os autores apresentam uma representação ontológica formal do código-fonte e dos artefatos de documentação, além de contemplar o preenchimento automático dessas duas ontologias através da análise de código-fonte e da mineração de texto.

Na Figura 4.1 é apresentada a arquitetura da abordagem proposta. A ontologia de software consiste em duas ontologias para a representação do código-fonte e de documentos do software. A infraestrutura semântica é formada pelas tecnologias bases da Web Semântica (RDF/OWL), Protégé (como editor de ontologias) e o Racer (sendo utilizado como mecanismo de inferência).

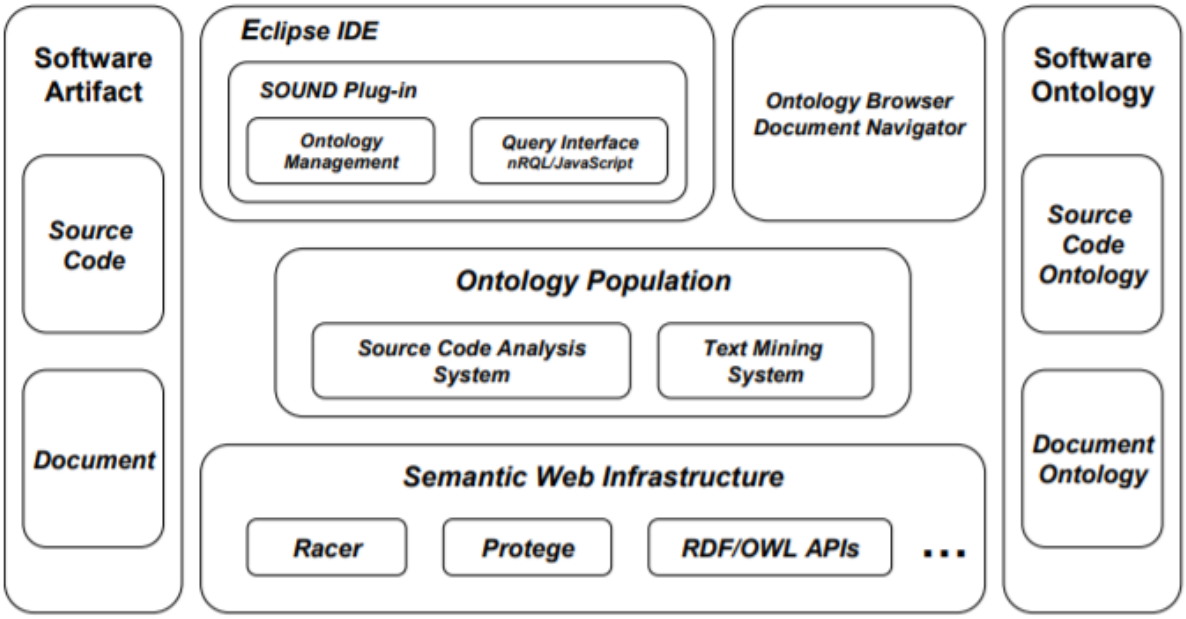


Figura 4.1: Arquitetura da abordagem utilizada por Witte et al. (2007).

O módulo de instância da ontologia é tratado em dois subsistemas: o primeiro para a extração dos dados referente ao código-fonte utilizando

o analisador JDT e o segundo subsistema para mineração de texto utilizando o framework GATE (General Architecture for Text Engineering). A interface de consulta é fornecida por um plugin integrando com a IDE Eclipse e as consultas são expressas na linguagem nRQL fornecida pelo Racer.

A ontologia de código-fonte especifica formalmente os principais conceitos do paradigma de programação orientada a objetos utilizando a linguagem Java, contemplando conceitos como: pacotes, classes, métodos e variáveis. O preenchimento da ontologia é baseado no JDT, um analisador de código-fonte fornecido pelo Eclipse. O analisador JDT extrai tokens do código-fonte gerando uma AST (Abstract Syntax Tree) e o subsistema de população da ontologia percorre a AST identificando os conceitos e suas relações para preenchimento da ontologia.

A ontologia de documentação consiste na descoberta de conceitos baseados em domínios de linguagens de programação, como: algoritmos, estrutura de dados, padrões de design e arquiteturas de software. O sistema executa várias etapas de pré-processamento para tokenização, divisão de frases, marcação de parte do discurso e divisão de frases substantivas. A utilização do OntoGazetteer contribui para anotar os tokens e suas correspondências na ontologia, por exemplo, a palavra "arquitetura" seria conceituada como uma classe na ontologia.

Como resultados foram testadas três abordagens: análise de segurança do código-fonte, conexões de rastreabilidade entre o código-fonte e a documentação e análise arquitetural.

A análise de segurança do código-fonte consiste em detectar vulnerabilidades de segurança no nível do código-fonte com base na ontologia. Neste cenário, foi realizada a identificação de vulnerabilidades relacionadas à causas de acessibilidade de objetos. A identificação de vulnerabilidades verifica a existência de objetos *public non-final*, ou seja, classes que possuem campos estáticos públicos. Os objetos públicos não finais são uma ameaça ao software, pois códigos maliciosos podem manipular o objeto para alterar o comportamento original do programa. Também, é possível validar se todos os atributos devem ser inicializados em construtores de classe.

A rastreabilidade envolvendo o código-fonte e a documentação do software permite que os mantenedores do projeto estabeleçam conexões através de consultas e raciocínios ontológicos. Deste modo, é possível realizar a correspondência de documentos a uma entidade do código-fonte, a fim de encontrar inconsistências entre a informação descrita no documento

de concepção do projeto e o desenvolvimento do código-fonte. A consulta identifica classes no código-fonte identificando a correspondência das entidades a partir da extração do texto do artefato de documentação, e, pode identificar métodos descritos na documentação que deveriam pertencer a uma determinada classe, mas na implementação do código-fonte esses requisitos não foram desenvolvidos.

No processo para a análise arquitetural, é possível consultar a quantidade de métodos e suas chamadas no sistema sobre as camadas do aplicativo. Deste modo, é possível observar no sistema analisado a quantidade de chamadas de métodos na camada de aplicação para a camada de domínio de forma direta, descartando a camada de controle.

- *Analyzing software with is iSPARQL* (Kiefer et al., 2007)

  Neste trabalho são utilizadas tecnologias da Web Semântica, em destaque o uso de ontologias, para enfrentar problemas relacionados a análise de sistemas de software. Os autores criaram um projeto denominado EvoOnt, que utiliza um conjunto de três ontologias para realizar a interoperabilidade de dados: ontologia de software, defeito e versionamento. Os autores também apresentam o uso do mecanismo de busca semântica nomeado de iSPARQL que é baseado em SPARQL contendo junções de similaridade.

  De acordo com os autores, a ontologia de software (Software Ontology Model) é um modelo independente de linguagem de programação, utilizada para a representação do código-fonte do paradigma orientado a objeto. A ontologia de defeitos (Bug Ontology Model) é baseada no sistema de rastreamento de defeitos do Bugzilla. Por fim, a ontologia que representa o versionamento de código (Version Ontology Model) especifica relações entre os arquivos do projeto e suas respectivas revisões e versões. Na Figura 4.2 é apresentado o modelo das três ontologias deste trabalho.

  Os experimentos deste trabalho foram realizados utilizando 206 versões do plugin "Eclipse Compare" para o ambiente de desenvolvimento Eclipse. Para o preenchimento das ontologias de software e de versionamento do código-fonte foi analisado automaticamente o repositório CVS do Eclipse. O preenchimento da ontologia de defeitos é realizado executando uma análise mais detalhada do CVS, obtendo as mensagens de cada *commit* e os respectivos rótulos de defeitos.

  Para realizar o experimento de evolução de código-fonte entre diferentes versões, foi realizada a comparação de todas as classes principais de uma nova versão com todas as outras classes da versão anterior.

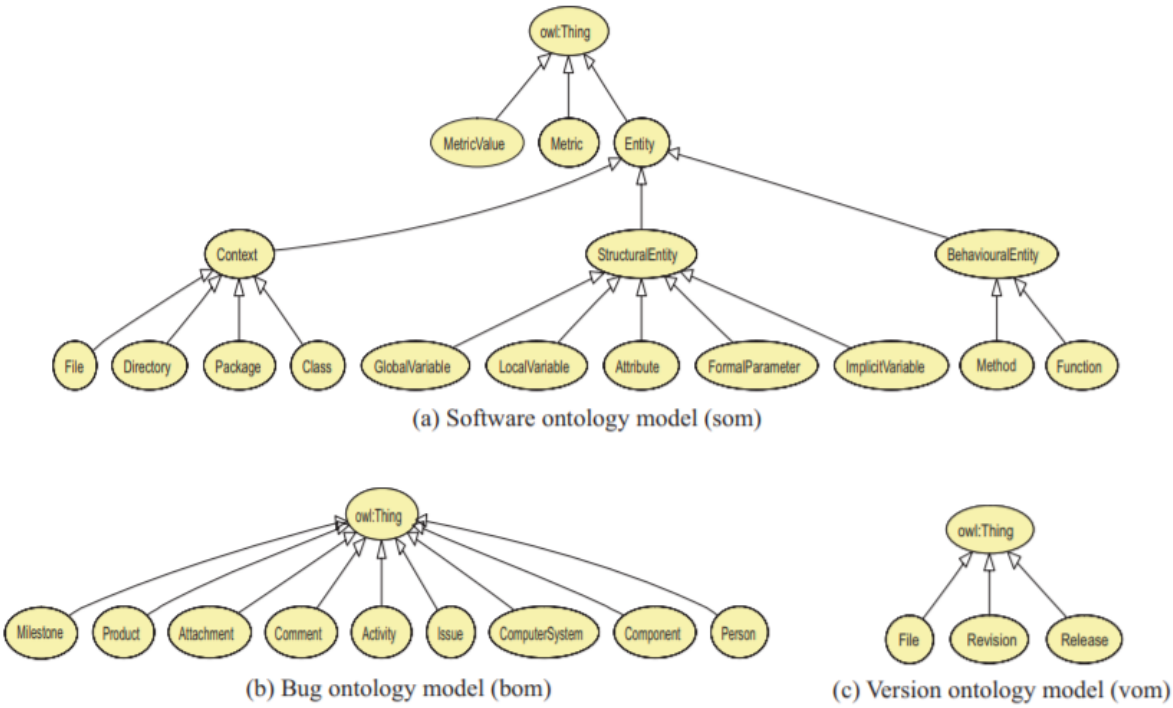


Figura 4.2: Visão geral das ontologias propostas por (Kiefer et al., 2007).

O experimento de refatoração aplicou duas buscas na ontologia de software para identificar pontos em que o código-fonte deve ser refatorado. A primeira busca tem como objetivo identificar classes que fazem referências diretas com outros objetos, pois ao efetuar a manutenção em uma classe nessa condição, todos os outros arquivos que a referenciam devem ser alterados. A segunda busca realiza a identificação de longas listas de parâmetros em métodos, sendo listados os itens candidatos para refatoração.

Os testes referentes as métricas de design de software realizam o cálculo de três métricas com o apoio da ontologia: número de métodos, número de atributos em uma classe e número de erros encontrados por classe.

O quarto experimento testou a capacidade de raciocínio ontológico realizando a busca de métodos “órfãos”, ou seja, métodos que não são invocados por nenhum outro método no projeto. Para esse experimento foi utilizado o mecanismo de inferência OWLMicro do framework Jena.

Por fim, os autores utilizam as métricas obtidas no experimento de design de software para calcular a densidade de defeitos e evolução do projeto de software, utilizando como razão o número de revisões e o número de erros encontrados.

- *Software bug ontology supporting semantic bug search on peer-to-peer networks* (Tran e Le, 2014)

  O trabalho de Tran e Le (2014) apresentam um sistema de busca de defeitos utilizando ontologias como forma de anotar semanticamente os dados dos relatórios de defeitos. O objetivo deste trabalho é explorar os relatórios de defeitos disponíveis em uma rede ponto a ponto (P2P), com o propósito de explorar defeitos semelhantes nesse ambiente, unificando vários relatórios de defeitos de diferentes sistemas em um único banco de dados. Os autores utilizam as tecnologias da Web Semântica em conjunto com estudos de sistemas distribuídos para efetuar buscas em redes P2P, utilizando querys em SPARQL. Neste trabalho, apenas será avaliado a metodologia pertinente a Web Semântica relacionada a ontologia.

  A ontologia de defeito proposta pelos autores é estendida de alguns vocabulários já existentes, além de outros estudos já realizados (Kiefer et al. (2007), Story (2012), BERGER Olivier (2010)). Porém, contendo propriedades que facilitam a descoberta entre as relações de relatórios de bugs. Na Figura 4.3 é representada a ontologia para a interoperabilidade de sistemas de rastreamento de defeitos, utilizando a linguagem UML para representação.

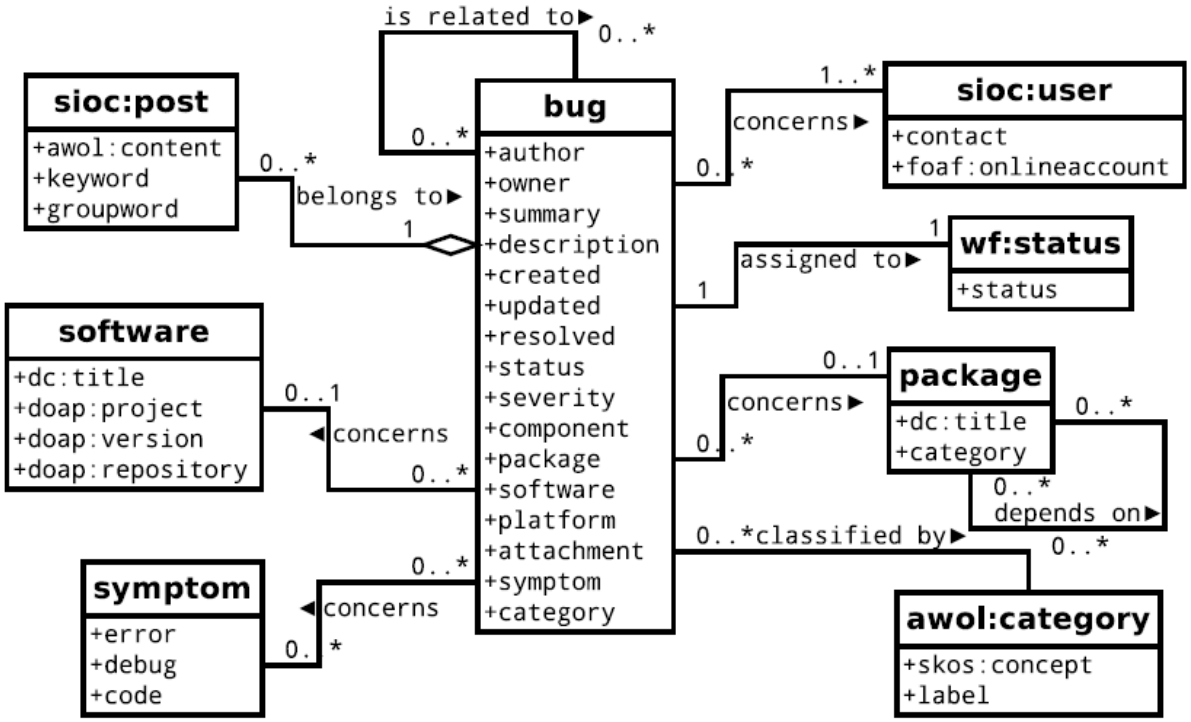


Figura 4.3: Ontologia de defeito proposta por (Tran e Le, 2014).

Para a classificação de defeitos, são consideradas as propriedades dos relatórios de defeitos que podem ser extraídas de forma direta para a aná-

lise de dependência de pacotes, classificação e relação dos componentes vinculados ao defeito relatado. A classificação de defeitos não considera apenas as propriedades já descritas no relatório de defeito, mas também, palavras-chave ou grupos de palavras-chave utilizando o método que mede a frequência de termos (TF: Term Frequency) que ocorrem no documento e a frequência inversa do documento (IDF: Inverse Document Frequency) que mede a importância de um termo.

A avaliação do método proposto foi realizada construindo uma busca para explorar os relatórios de defeitos obtidos em diferentes sistemas de rastreamentos de defeitos, o método de pesquisa explora as características presentes nos relatórios de defeitos e os métodos de classificação TF x IDF para recuperar relatórios de defeitos semelhantes na rede P2P. As consultas realizadas na ontologia utilizam duas métricas: relatórios de defeitos idênticos e relatórios de defeitos semelhantes. A primeira métrica filtra os defeitos que estão presentes no mesmo pacote e em pacotes dependentes. A segunda métrica verifica as palavras-chave correspondentes e sua frequência para identificar semelhanças. Na Figura 4.4 é apresentado os dados obtidos de diferentes sistemas de rastreamentos de defeitos unificados com a ontologia proposta.

BTS Site	BTS System	Number of Bugs
bugs.debian.org	Debian BTS	500000
bugs.eclipse.org	Bugzilla	270000
bugs.gentoo.org	Bugzilla	190000
bugzilla.mozilla.org	Bugzilla	300000
bugzilla.redhat.com	Bugzilla	200000
issues.asterisk.org	Mantis	20000
bugs.scribus.net	Mantis	10000
bugs.launchpad.net	Launchpad	950000

Figura 4.4: Quantidade de defeitos extraídos utilizando a ontologia proposta por (Tran e Le, 2014).

Para a avaliação de defeitos idênticos na rede é realizado um conjunto de consultas no banco de dados de defeitos que contenham sobreposição de conjuntos de forma intencional, por exemplo, um nó possui relatórios de defeitos do Mozilla variando entre os identificadores 50.000 a 150.000 e outro nó variando de 100.000 a 200.000. O método de pesquisa para defeitos semelhantes é executado de maneira parecida nos bancos de dados Mozilla e Debian, porém a quantidade de relatórios de defeitos semelhantes recuperados aumenta à medida que as consultas possuem menos palavras-chave.

Como resultado, na Figura 4.5 é apresentada a taxa de precisão relaci-

onada com a taxa de recuperação para relatórios de defeitos idênticos (esquerda) e a taxa de precisão relacionada com a taxa de recuperação para relatórios de defeitos semelhantes (direita).

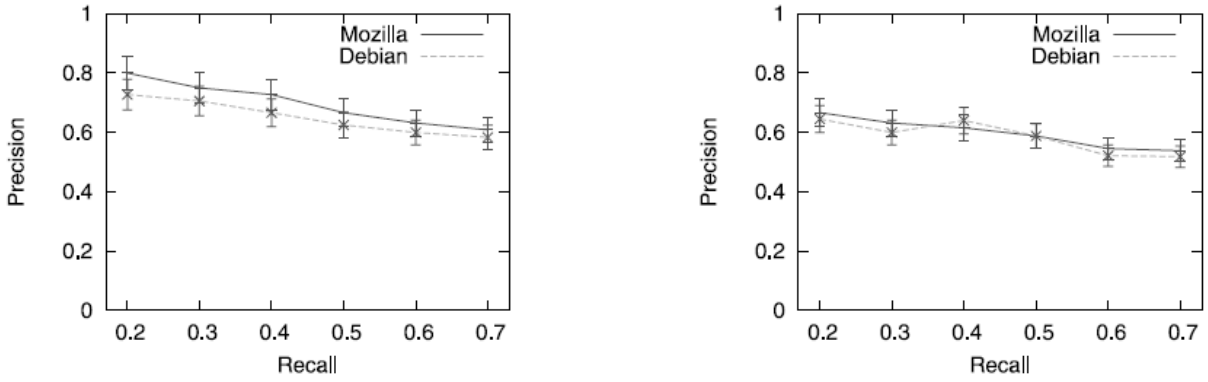


Figura 4.5: Precisão de pesquisa de defeitos idênticos x semelhantes (Tran e Le, 2014).

4.2 Ontologia do Código-Fonte

- *Providing a Source Code Security Analysis Model Using Semantic Web Techniques* (EkramiFard e Kahani, 2015)

  Neste trabalho os autores apresentam o uso de tecnologias da Web Semântica para detectar falhas de segurança no código-fonte. Deste modo, são utilizados alguns padrões de falhas de segurança e convertidos em consultas semânticas na ontologia do código-fonte utilizando a linguagem SPARQL. Na Figura 4.6 é apresentada a arquitetura do modelo que consiste em duas etapas:

  - Análise: O analisador obtém o código-fonte e gera uma AST (Abstract Syntax Tree) produzindo triplas RDF que são armazenadas em um arquivo com a extensão n3.
  - Raciocínio: Os padrões de falhas são convertidos em consultas SPARQL e são executados na ontologia gerada na etapa anterior.

  A avaliação realizada pelos autores consiste em pesquisar os modelos de falhas: *ConstantDBPass*, *MReInternalArray* e *SecuritySer* implementados em consultas SPARQL na ontologia e comparar o resultado das detecções desse padrões com as detecções realizadas na ferramenta FindBugs. Na Figura 4.7 é demostrado os resultados experimentais dessa avaliação.

  Na Figura 4.8 é apresentada uma consulta SPARQL gerada para detectar o padrão *SecuritySer*, que identifica classes que implementam a interface

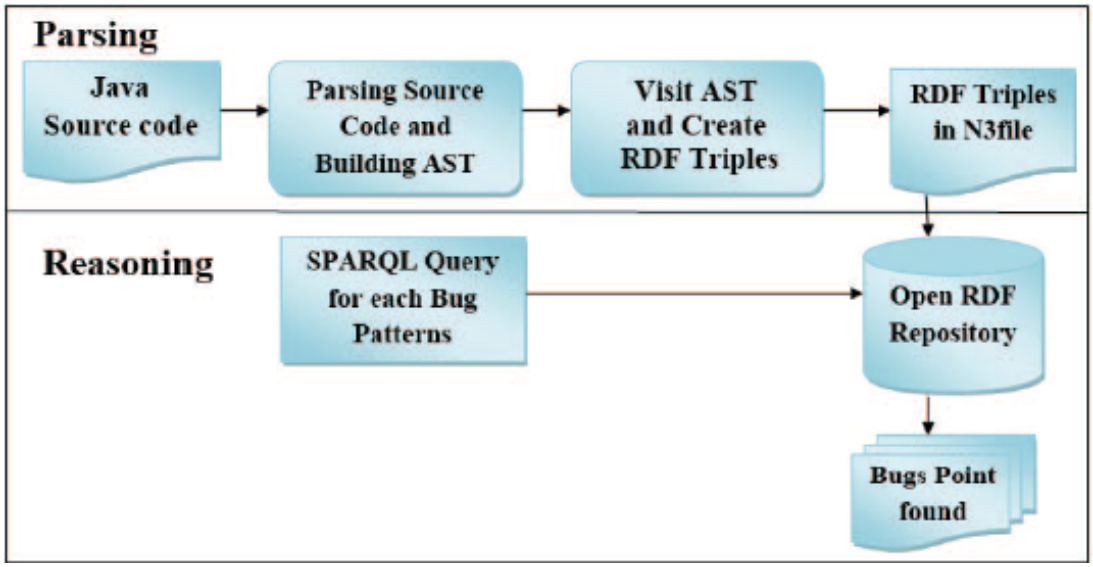


Figura 4.6: Arquitetura proposta para geração e uso da ontologia (EkramiFard e Kahani, 2015).

Bug Pattern Name	Proposed Model	FindBugs
ConstantDBPass	√	√
MReInternalArray	√	×
SecuritySer	√	×

√. Detect
×. Not Detect

Figura 4.7: Resultado de detecção comparado com a ferramenta FindBugs (EkramiFard e Kahani, 2015).

```
SELECT * WHERE {
        ?cls  rdf:type   j2rdf:class.
        ?cls  j2rdf:Implements  ?intf .
        ?intf  rdf:type   j2rdf:interface .
        ?intf  j2rdf:interface  "Serializable".}
```

Figura 4.8: Consulta SPARQL para detecação do padrão SecuritySer (EkramiFard e Kahani, 2015).

"java.io.Serializable". A consulta possibilita que sejam efetuadas buscas mais precisas do que apenas a análise sintática do código-fonte, assim

como são realizadas pela ferramenta FindBugs.

- *CodeOntology: RDF-ization of Source Code* (Atzeni e Atzori, 2017)

  Neste trabalho Atzeni e Atzori (2017) apresentam os resultados do desenvolvimento de uma ontologia de código-fonte para a linguagem Java, denominada CodeOntology. O trabalho inclui estudos de modelagem de dados (RDF), gerenciamento de dados e consultas (JENA e SPARQL). O projeto apresenta duas contribuições: a primeira sendo uma ontologia que fornece a representação formal de linguagens de programação orientadas a objetos, focado na linguagem Java, e, a segunda contribuição, um analisador responsável por serializar o código-fonte Java em triplas RDF.

  O analisador atua na extração de informações estruturais do código-fonte, como: hierarquias de classes, métodos e construtores. Deste modo, o conjunto dos artefatos desenvolvidos neste projeto possibilita a execução de consultas semânticas sobre o código-fonte utilizando a linguagem SPARQL.

  A ontologia desenvolvida é composta por 65 classes, 86 propriedades de objetos e 11 propriedades de dados. O analisador que realiza a serialização do código-fonte em triplas RDF atua em três etapas: na primeira etapa é realizado o download de todas as dependências do projeto (atualmente suporta os projetos desenvolvidos em Maven e Gradle), em seguida gera uma AST (Abstract Syntax Tree) do código-fonte e, por fim, inicia o processamento para armazenar as triplas RDF na ontologia. Na Figura 4.9 é apresentado o processo de serialização do código-fonte em triplas RDF e na Figura 4.10 é apresentado a ontologia do projeto.

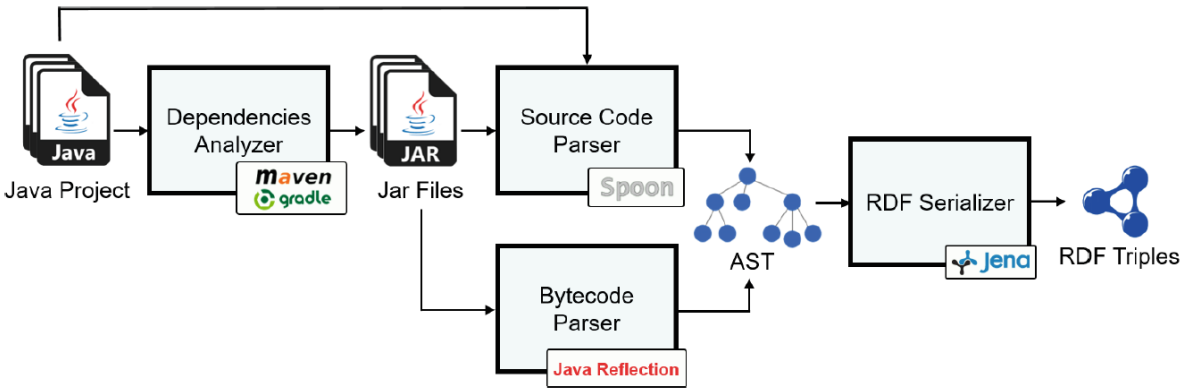


Figura 4.9: Processo de serialização do código-fonte em triplas RDF (Atzeni e Atzori, 2017).

```java
package org.codeontology;

public class Example {
  /** Prints a "hello world" message to the standard output */
  public static void main(String[] args) {
    System.out.println("Hello CodeOntology!");
  }
}
```

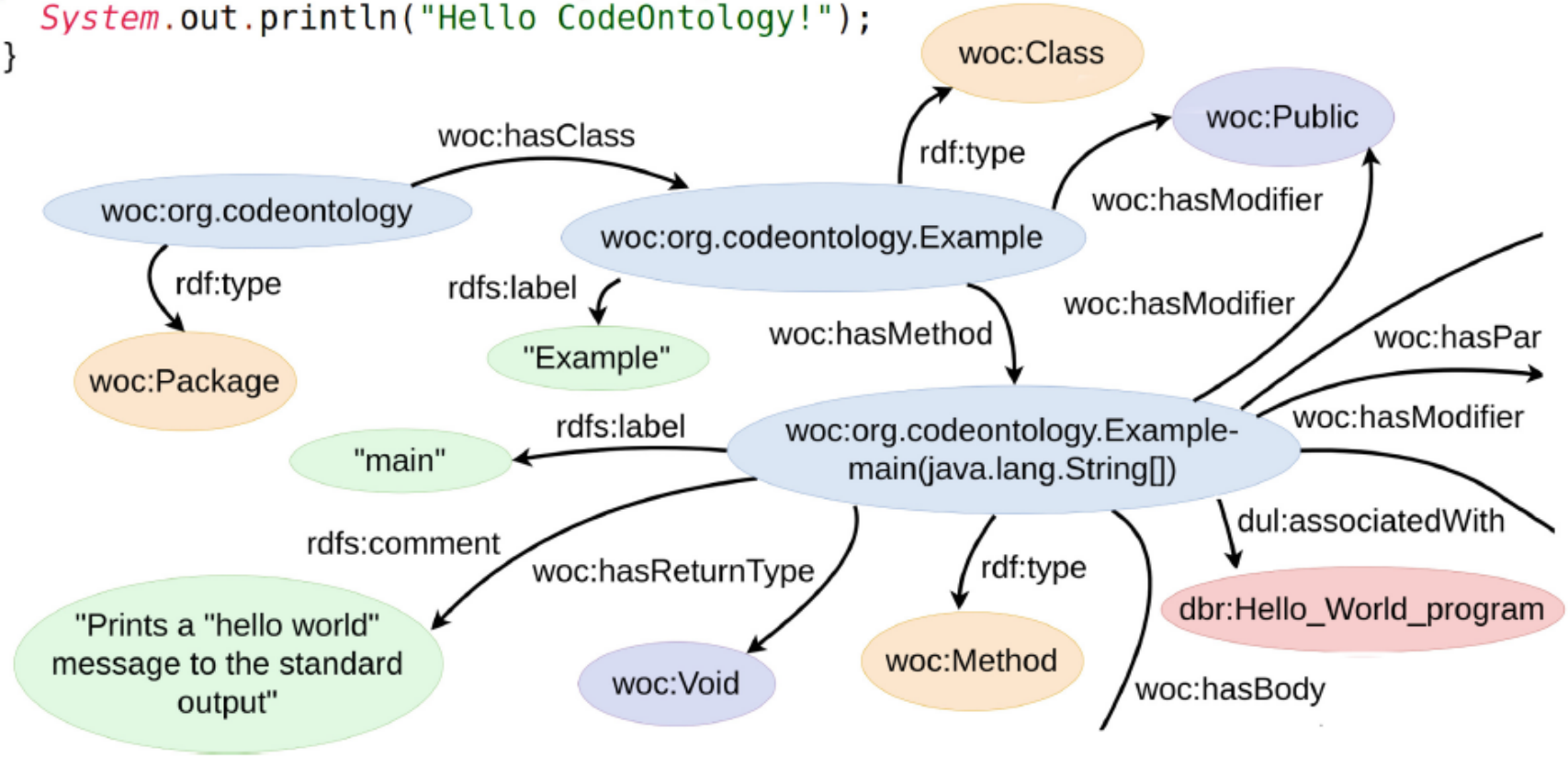


Figura 4.10: Serialização RDF produzida para o método de exemplo (Atzeni e Atzori, 2017).

Além do projeto OpenJDK 8, o trabalho testou outros 20 projetos desenvolvidos na linguagem Java obtidos de repositórios aleatórios do GitHub. Na Figura 4.11 é apresentada como resultado o tempo necessário para o download das dependências do projeto, o tempo decorrido para análise do código-fonte, o tempo de processamento do arquivo JAR e a quantidade de triplas RDF extraídas de cada projeto. Para os registros que não possuem dados na coluna "source code", não foi realizado a análise do código-fonte, pois houve falha no processo de geração da AST.

Download	Source code	JAR	Total time	RDF triples
18.5	3.0	0.1	21.6	4336
30.3	–	4.2	34.4	544744
20.2	–	2.5	22.7	344465
15.3	–	0.2	15.5	2496
129.3	21.8	6.0	157.2	626607
94.6	38.3	2.4	135.3	212258
18.2	–	0.1	18.3	2598
0.1	–	1.4	1.6	90071
78.4	–	0.1	78.4	2597
95.6	–	5.3	100.9	505262
258.1	9.9	162.9	430.9	9152328
2950.5	99.7	216.7	3267.1	17059138
171.8	–	0.2	172.0	2499
53.8	–	0.2	54.0	2496
47.0	–	6.9	54.0	561580
121.1	–	2.4	123.4	95376
140.8	93.9	3.5	238.1	171267
78.3	–	0.1	78.6	4992
34.2	–	10.9	45.2	1101273
26.4	–	1.2	27.6	26212
4382.5	266.6	427.3	5076.8	30512595

Figura 4.11: Tempo para o processo de serialização do código-fonte em 20 projetos diferentes (Atzeni e Atzori, 2017).

- *OOC-O: a Reference Ontology on Object-Oriented Code* (Aguiar et al., 2019)

  Neste trabalho Aguiar et al. (2019) os autores apresentam uma ontologia de referência para a representação do paradigma de programação orientado a objeto (OOC-O), fornecendo uma camada de interoperabilidade e unificando a representação de diferentes linguagens de programação deste paradigma em um modelo semântico.

  A ontologia OOC-O pode ser aplicada em diversos contextos, porém a sua construção é focada no cenário de programação poliglota, onde em um mesmo projeto estão presentes diferentes linguagens de programação.

Para a construção da ontologia OOC-O os autores consideraram as linguagens de programação Smalltalk, Eiffel, C++, Java e Python. Os autores desenvolveram a ontologia OOC-O de forma modular para favorecer a sua reutilização e evolução em outros contextos.

A ontologia OOC-O é composta por três módulos:

- ***Core Module***: O módulo principal descreve os conceitos fundamentais do paradigma de programação orientado a objetos.
- ***Class Module***: Este módulo descreve os conceitos e relações fundamentais do paradigma de programação orientado a objetos.
- ***Class Member Module***: Neste módulo são descritos os conceitos e relações fundamentais entre classes, métodos, atributos e variáveis do paradigma de programação orientado a objetos.

A ontologia possui 43 classes e 20 propriedades. Na Figura 4.12 é apresentado o módulo principal que compõe a ontologia OOC-O. As entidades em amarelo fazem parte da ontologia desenvolvida pelos autores, já as outras entidades são extensões de outras ontologias que compõem a ontologia OOC-O.

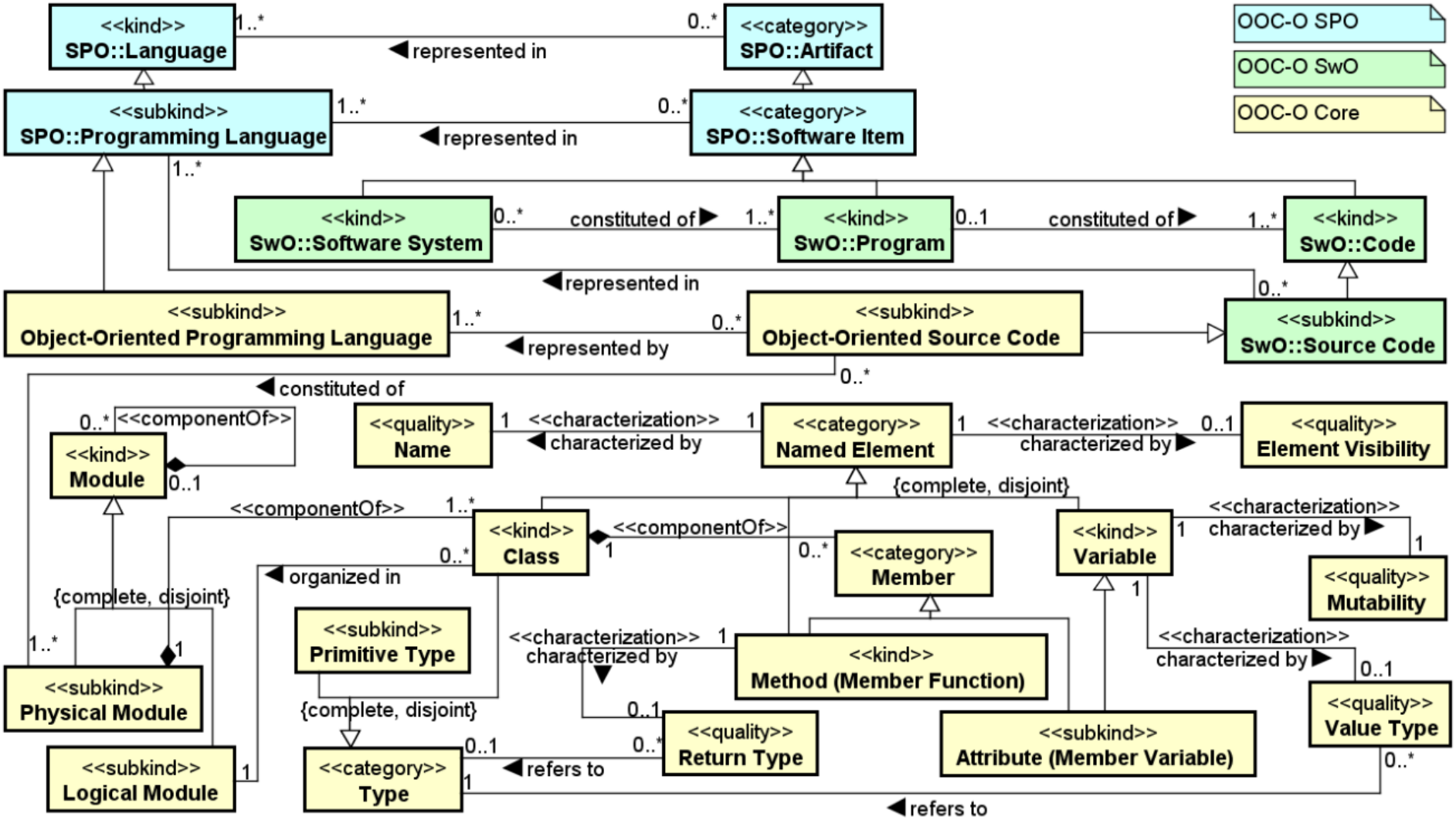


Figura 4.12: *Core Module* da ontologia OOC-O (Aguiar et al., 2019).

4.3 Processo de Localização de Defeitos

- *Leveraging textual properties of bug reports to localize relevant source files* (Gharibi et al., 2018)

  O trabalho de Gharibi et al. (2018) tem como objetivo propor uma abordagem de localização de defeitos considerando várias propriedades extraídas do relatório de defeito e arquivos de código-fonte, além das relações dos artefatos alterados com base no histórico de relatórios de defeitos. Nesse projeto os autores consideram a utilização de técnicas de recuperação da informação e categorização de texto para melhorar a eficiência do processo de localização de defeitos.

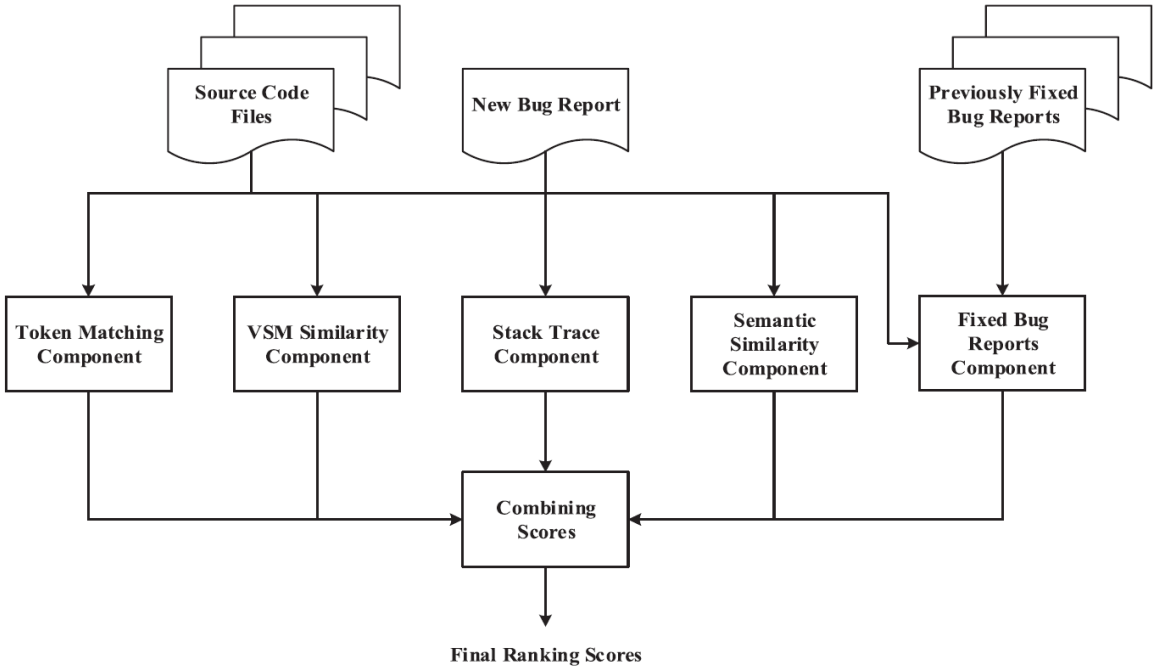


Figura 4.13: Arquitetura do modelo para localização de defeitos (Gharibi et al., 2018).

A Figura 4.13 apresenta a arquitetura do projeto proposto por este artigo. Após a fase de análise e pré-processamento do relatório de defeito os cinco componentes identificados como correspondência de token, similaridade VSM, rastreamento de pilha, similaridade semântica e relatórios de defeitos corrigidos geram uma pontuação para cada artefato do código-fonte encontrado. Por fim, essas pontuações são combinadas para obter uma classificação final, ordenando os arquivos de código-fonte relevantes em relação a cada relatório de defeito reportado.

Após o processamento das propriedades extraídas do relatório de defeito e o histórico de modificações dos artefatos de código-fonte, é atribuída

uma pontuação para cada artefato encontrado e posteriormente realizado uma classificação final. A classificação final ordena os arquivos de código-fonte relevantes em relação a cada relatório de defeito reportado.

Para avaliar os resultados dessa abordagem foram avaliados três projetos de software: AspectJ, SWT e ZXing, além de comparar com os resultados de outras quatro ferramentas de localização de defeitos: BugLocator, BLUiR, BRTracer e AmaLgam.

Os resultados mostram que a abordagem proposta pelos autores pode classificar artefatos de código-fonte apropriados para alteração em 52,5% dos defeitos reportados e relacionar o artefato em uma lista com 10 artefatos em 78,2% dos defeitos reportados. Comparado com outras ferramentas relacionadas com o processo de localização de defeitos, as métricas são superiores, sendo respectivamente 37,8%, 25,4%, 16,8% e 12,5% em relação as ferramentas BugLocator, BLUiR, BRTracer e AmaLgam.

- *Bug Localization Approach using Source Code Structure with Different Structure Fields* (Swe e Oo, 2018)

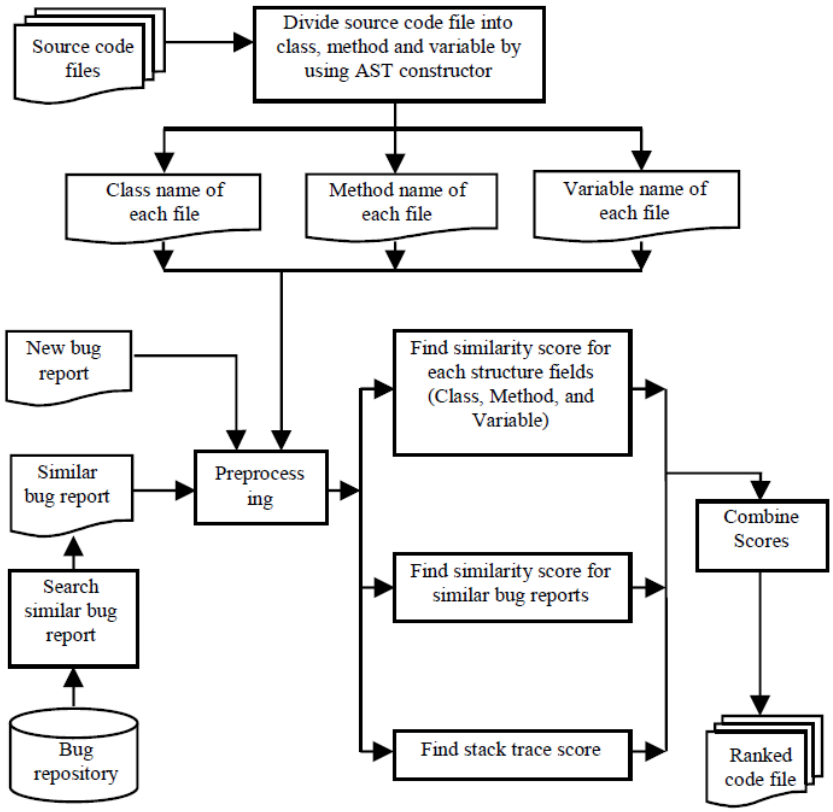


Figura 4.14: Arquitetura do modelo para localização de defeitos proposta (Swe e Oo, 2018).

Neste trabalho Swe e Oo (2018) os autores apresentam um modelo de localização de defeitos que difere das abordagens que utilizam os artefatos

de código-fonte como unidades, o que geralmente pode causar muitos ruídos na classificação de pontuação quando o arquivo possui muitas linhas de código. Deste modo, o modelo proposto considera o uso das informações estruturais do relatório de defeito para analisar a similaridade com os artefatos de código-fonte e o rastreamento de pilha.

A Figura 4.14 apresenta a arquitetura do projeto proposto pelos autores, o processo apresentado considerado como entrada o novo relatório de defeito reportado, os arquivos de código-fonte e relatórios de defeitos já corrigidos. Ao obter os artefatos de código-fonte essa abordagem realiza a extração de três propriedades relevantes do código-fonte: nome da classe, métodos e variáveis.

O cálculo de ranqueamento considera a similaridade de cada propriedade extraída do código-fonte, atribuindo pontuações utilizando a similaridade de cosseno. Os relatórios de defeitos também são pontuados de acordo com sua similaridade, e, por fim, é atribuída a pontuação do rastreamento de pilha. Combinando todas as três pontuações, é realizada a classificação dos possíveis artefatos candidatos a alteração.

Para avaliar os resultados dessa abordagem, foram avaliados três projetos de software: AspectJ, SWT e Eclipse.

Para o projeto Eclipse foi analisado 3.075 relatórios de defeitos, o sistema localizou com sucesso 18,5% artefatos de código-fonte, em 40,4% o artefato ficou listado em um dos cinco possíveis artefatos e 49,8% em uma entre os dez possíveis artefatos.

O projeto AspectJ relacionou 286 defeitos, tendo como resultado da localização 25,5%, 43% e 52% dos arquivos de código-fonte alterados classificados em primeiro, entre os cinco ou entre os dez principais resultados, respectivamente.

O projeto SWT contendo 98 relatórios de defeito, gerou como resultado 29,5% dos arquivos de código-fonte alterados classificados em primeiro, 59,1% e 68,3% entre os cinco ou entre os dez principais resultados, respectivamente.

4.4 Considerações Finais

No trabalho de Witte et al. (2007) não são apresentadas as representações ontológicas dos artefatos de documento e de código-fontes. A metodologia de extração do código-fonte utilizando AST contribui para o processo de instância das classes da ontologia, porém não é possível avaliar se a representação deste domínio é completa o suficiente para a elaboração de consultas semânticas e

inferências na ontologia. A relação para a busca de conexões de rastreabilidade entre o código-fonte e a documentação não são tão precisas, pois com a implementação do código-fonte não é possível identificar uma relação com o requisito documentado. O nível de vocabulário da ontologia de código-fonte é primordial para que seja possível a criação de consultas SPARQL robustas para a extração de conhecimento do código-fonte.

O trabalho de Kiefer et al. (2007) apresentam um projeto denominado Evo-Ont, que utiliza um conjunto de três ontologias para realizar a interoperabilidade de dados: ontologia de software, defeito e versionamento. Aperfeiçoando e complementando algumas lacunas presentes no trabalho Witte et al. (2007), porém as ontologias criadas para a representação dos domínios são muito superficiais, ou seja, não possuem todas as características de uma ontologia, apresentando apenas uma taxionomia de alguns elementos do domínio sem regras e relacionamento. Os autores também não especificam com detalhes a metodologia para preenchimento da ontologia de software, assim como as ontologias de versionamento e defeito.

O trabalho de Tran e Le (2014) propõe uma ontologia de defeitos unificada, que é estendida de alguns vocabulários já existentes para combinar relatórios de defeitos presentes em diferentes sistemas de rastreamento de defeitos. Essa proposta contempla uns dos objetivos do uso de ontologias, que é a interoperabilidade entre diferentes sistemas que atuam em um mesmo domínio. Os próprios autores concluem que a avaliação de relatórios de defeitos similares usando a metodologia de classificação TF x IDF ainda está incompleta, pois os bancos de dados de defeitos possuem um escopo amplo e permanecem praticamente inexplorados. Neste cenário, torna-se um domínio muito propicio para o estudo de aplicações da Web Semântica.

No trabalho de EkramiFard e Kahani (2015) os autores buscam utilizar as tecnologias da Web Semântica para detectar falhas de segurança no código-fonte, convertendo padrões de falhas de segurança em consultas semânticas usando SPARQL. Os autores não disponibilizam a representação da ontologia, mas o processo consiste na geração de uma ontologia de código-fonte utilizando AST, que é comum em outros trabalhos. Ao efetuar pesquisas de falhas utilizando consultas semânticas em uma ontologia de código-fonte, pode-se concluir que são detectados com muito mais eficiência quando comparados a métodos que fazem apenas a busca sintática no código-fonte. Para melhorar a eficiência da busca, pode ser desenvolvida uma ontologia para a descrição das regras e relacionamentos de cada padrão de falha de segurança e obtidos tais dados de uma base de dados de vulnerabilidades.

Os autores do trabalho Atzeni e Atzori (2017) apresentam uma ontologia para a representação do código-fonte para a linguagem Java, composta por

65 classes, 86 propriedades de objetos e 11 propriedades de dados. O processo de geração da ontologia consiste em analisar e realizar o download de todas as dependências do projeto, gerar uma AST do código-fonte utilizando a biblioteca *Spoon* e serializar o código-fonte em triplas RDF.

O trabalho de Aguiar et al. (2019) possuem um objetivo semelhante ao apresentado por Atzeni e Atzori (2017), porém o objetivo é apenas o desenvolvimento de uma ontologia de referência para a representação do paradigma de programação orientado a objeto. A ontologia apresentada por Aguiar et al. (2019) pode ser aplicável em diversos contextos, porém os autores enfatizam que "a ontologia está sendo construída no contexto da programação poliglota, ou seja, diferentes linguagens de programação com diferentes construções combinadas em um mesmo projeto de desenvolvimento de software".

O processo proposto por Atzeni e Atzori (2017) permite o desenvolvimento de novas pesquisas ainda inexploradas utilizando uma ontologia para a representação formal do código-fonte em uma estrutura semântica. A ontologia pode ser usada como apoio para melhorar diversos processos, que com base em análises estatísticas, buscam identificar a relação entre artefatos gerados no processo de engenharia de software com os artefatos do código-fonte.

Nos trabalhos de Gharibi et al. (2018) e Swe e Oo (2018) são apresentados modelos para o processo de localização de defeitos, esses modelos buscam utilizar algoritmos para identificar a similaridade semântica e a relação entre os relatórios de defeitos e os artefatos de código-fonte. No trabalho Gharibi et al. (2018) são utilizados cinco diferentes componentes para atribuir uma pontuação de classificação para os artefatos candidatos a manutenção, já o trabalho Swe e Oo (2018), difere na extração de três propriedades relevantes do código-fonte: nome da classe, métodos, e as variáveis, calculando a similaridade de cada propriedade extraída do código-fonte e atribuindo as respectivas pontuações. Desta forma, o artefato de código-fonte não é tratado como uma unidade, diminuindo ruídos causados na classificação da pontuação de cada artefato.

Na Tabela 4.1 são elencadas as principais características dos trabalhos relacionados apresentados. É possível analisar que na maioria dos projetos são utilizadas ontologias para a representação semântica dos artefatos e para a modelagem semântica do código-fonte. Mesmo com a maioria dos trabalhos relacionados beneficiando-se do uso de ontologias, os processos relacionados a localização de defeitos não se beneficiam da modelagem semântica.

Neste cenário os trabalhos relacionados contribuíram para identificar as lacunas de pesquisa nos modelos de localização de defeitos e técnicas de outros trabalhos que quando utilizadas em conjunto podem produzir benefícios para estes modelos.

Tabela 4.1: Comparação das principais características dos trabalhos relacionados

Trabalhos relacionados	Localização de defeitos	Uso de ontologias	Modelagem semântica do código-fonte
Witte et al. (2007)		X	X
Kiefer et al. (2007)		X	X
Tran e Le (2014)		X	
EkramiFard e Kahani (2015)		X	X
Atzeni e Atzori (2017)		X	X
Aguiar et al. (2019)		X	
Gharibi et al. (2018)	X		
Swe e Oo (2018)	X		

CAPÍTULO 5

LOBUS-OWL

Neste capítulo é apresentado o modelo para localização de defeitos apoiado por ontologia, denominado LOBUS-OWL. A Figura 5.1 apresenta a formação do acrônimo LOBUS-OWL. Na Seção 5.1 é abordado a metodologia geral do projeto e a arquitetura de alto nível. Os módulos da arquitetura são descritos com detalhes nas demais seções.

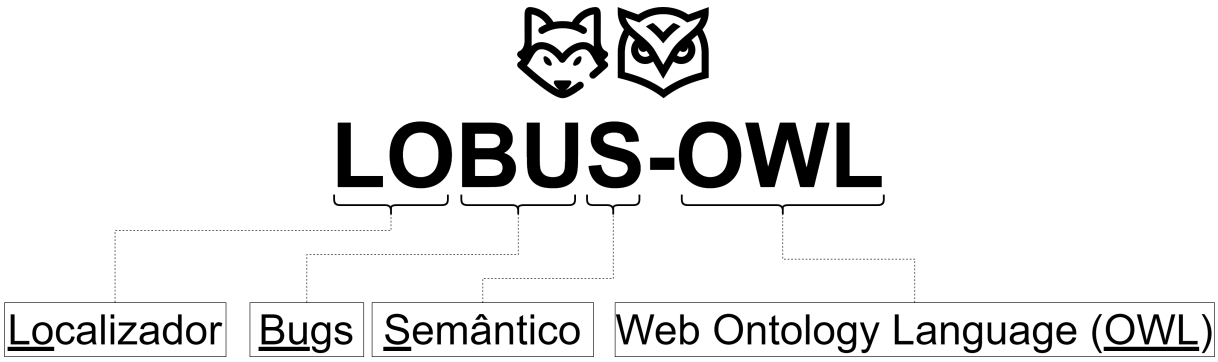


Figura 5.1: Acrônimo LOBUS-OWL (Elaborado pelo Autor).

5.1 Metodologia

O trabalho inicia como uma pesquisa exploratória, com a revisão da literatura que auxiliou na obtenção de conhecimento para a construção e a elaboração do modelo proposto na pesquisa. Também, caracteriza-se como uma pesquisa descritiva e aplicada, pois foram obtidas informações na literatura para a identificação do objeto de pesquisa. Por fim, o desenvolvimento do modelo de localização de defeitos identificando e classificando os artefatos de código-fonte candidatos relacionados com o relatório de defeitos reportado.

Na Figura 5.2 é apresentado a arquitetura de alto nível do projeto LOBUS-OWL. Este projeto atua na automatização do processo de localização de defeitos, considerando a estrutura arquitetônica e semântica do código-fonte. A arquitetura está divida em quatro módulos detalhados nas subseções seguintes.

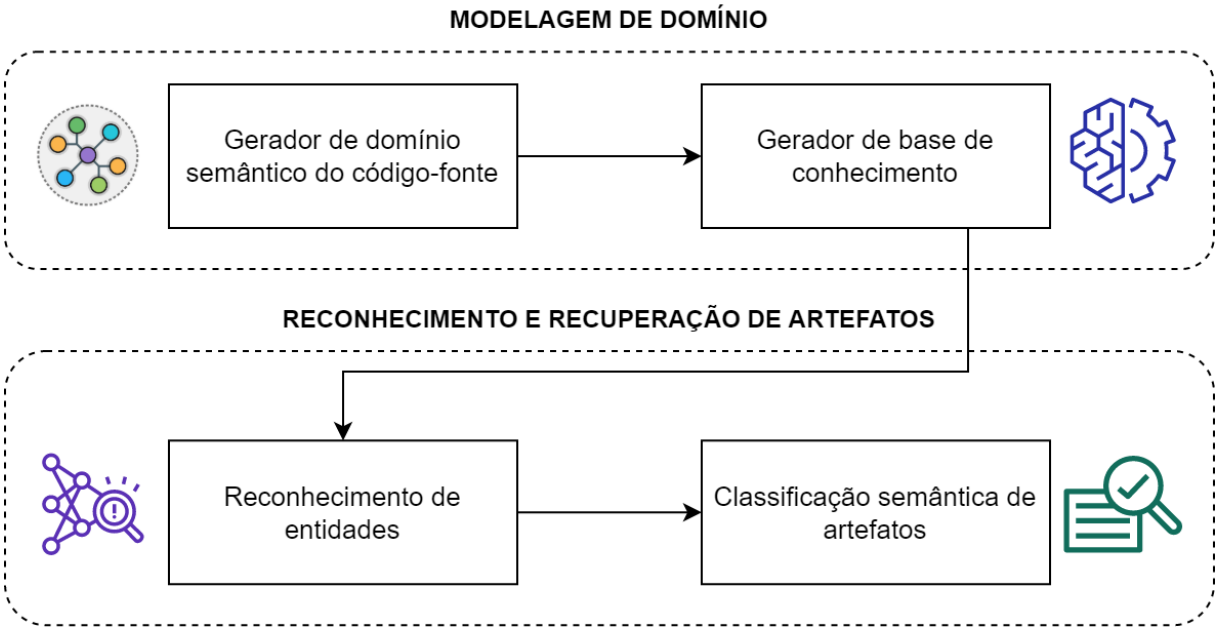


Figura 5.2: Arquitetura do modelo para localização de defeitos (Elaborado pelo Autor).

5.2 Gerador de domínio semântico do código-fonte

O módulo gerador de domínio semântico do código-fonte é responsável por realizar a conversão do código-fonte de projetos codificados utilizando o paradigma de programação orientado a objeto e escritos na linguagem C Sharp (C#). O objetivo deste módulo para o processo de localização de defeitos é utilizar os benefícios da modelagem ontológica para que seja possível aplicar a semântica das entidades do projeto na localização de defeitos. Para a extração das entidades do código-fonte são utilizadas as bibliotecas do compilador Roslyn (Microsoft, 2021) e MsBuild (Microsoft, 2022f).

Para a análise dos resultados foram utilizados os projetos AutoMapper, MsBuild, EfCore, AspNetCore, MQTTnet e NLog, obtidos de repositórios abertos do GitHub. O critério para a escolha dos projetos é detalhado posteriormente no Capítulo 6. Na Figura 5.3 é apresentado a arquitetura deste módulo.

A ontologia OOC-O (Aguiar et al., 2019) é utilizada para representar a semântica do código-fonte do paradigma de programação orientado a objeto. A ontologia representa as entidades semânticas do projeto em tempo de compilação e pode representar qualquer projeto de código-fonte que esteja codificado

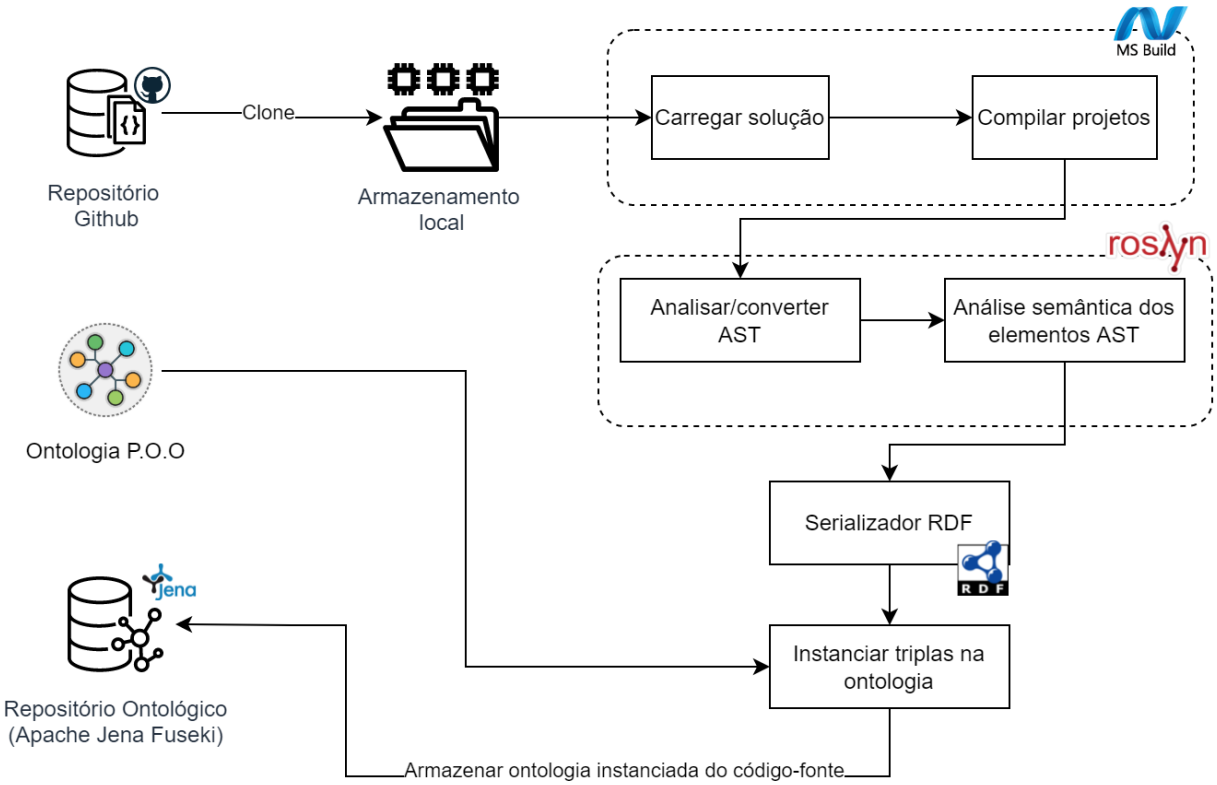


Figura 5.3: Módulo gerador de domínio semântico do código-fonte (Elaborado pelo Autor).

utilizando o paradigma de programação orientado a objeto.

Para este projeto foram realizadas algumas modificações na ontologia para detalhar a semântica e as relações de algumas entidades identificadas durante o desenvolvimento, que são particulares da linguagem C Sharp. A Figura 5.4 apresenta parte da ontologia onde novas entidades foram adicionadas para contemplar a arquitetura dos projetos codificados na linguagem C Sharp. Foram adicionadas as seguintes entidades:

- Solution: A solução pode possuir vários programas.
- Program: Programa ao qual pertence o arquivo de código-fonte físico.
- File: Arquivo físico onde é salvo o código-fonte, constituído pelo respectivo módulo da aplicação.
- Mutator_Method: Método que fornece uma interface entre os dados internos do objeto e o mundo externo, ou seja, permite o acesso a variáveis de instância privada de um objeto. No C Sharp são conhecidos como propriedades da classe.

O gerador de domínio semântico do código-fonte inicia obtendo todos os arquivos do projeto no repositório do GitHub e realizando a persistência desses

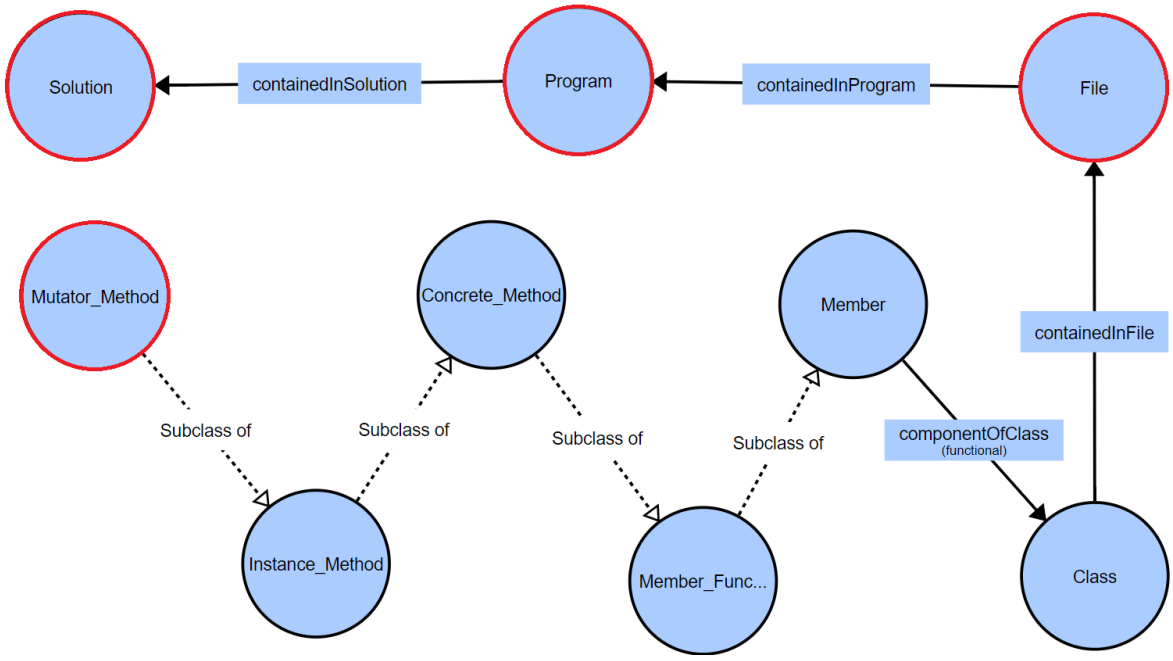


Figura 5.4: Novas entidades da ontologia OOC-O em destaque (Elaborado pelo Autor).

arquivos em um diretório do sistema. O gerador obtém o caminho do arquivo da solução do projeto no diretório do sistema e inicia o carregamento de seus projetos. Apenas são considerados arquivos de extensão ".cs" e as dependências necessárias para a compilação exigidas pela biblioteca MsBuild. Nessa etapa é possível realizar o carregamento de uma ou mais soluções, projetos ou um conjunto de arquivos independentes para análise.

Ao realizar o carregamento do projeto inicia-se o processo de compilação, esse processo também utiliza a biblioteca MsBuild, responsável por realizar a validação de dependências e a compilação das soluções, projetos ou arquivos. Após a compilação realizada na etapa anterior, é possível obter a estrutura da árvore sintática abstrata (AST) do código-fonte para a análise. Nessa etapa são utilizados alguns recursos do compilador Roslyn do C Sharp para poder realizar a extração da AST de cada arquivo de extensão ".cs".

A Figura 5.5 apresenta a AST gerada para a declaração do método "Validade", apresentado no trecho do código-fonte 5.1 do projeto AutoMapper. Na Figura 5.5 os elementos em azul representam os nós de construções da sintaxe, como: declarações, operadores, expressões e etc. Os elementos em verde são os tokens que incluem identificadores, palavras-chave e caracteres especiais. Os elementos em branco ou cinza representam informações adicionais sobre o nó, como: espaços, caracteres de fim de linha, quebra de linha e etc.

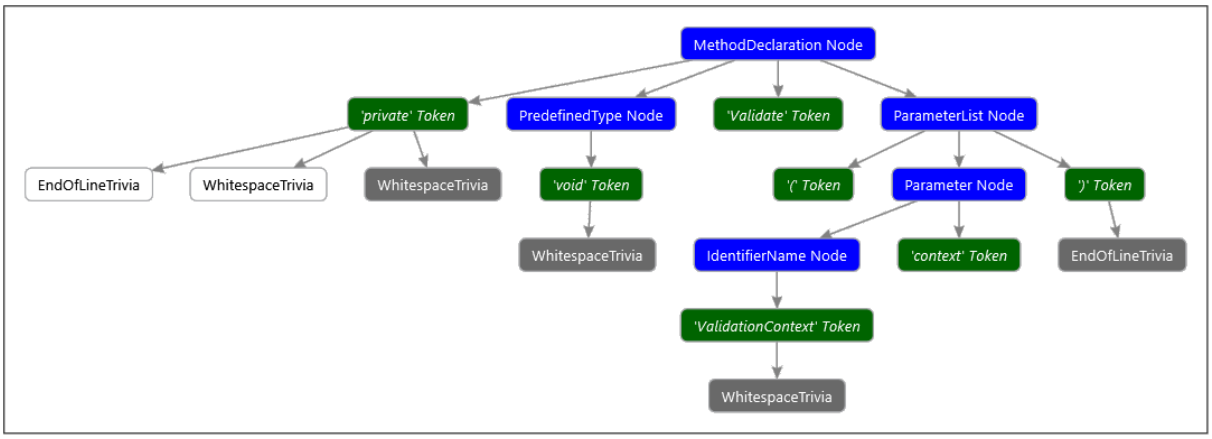


Figura 5.5: AST referente a declaração do método *Validate* do projeto AutoMapper (Elaborado pelo Autor).

```
1 private void Validate(ValidationContext context)
2 {
3     foreach (var validator in _validators)
4     {
5         validator(context);
6     }
7 }
```

Listing 5.1: Método *Validate* do projeto AutoMapper

Com a AST do código-fonte gerada, a próxima etapa o gerador semântico percorre todos os elementos da AST para obter as informações semânticas dos objetos, tais como: namespaces, nome de entidades, tipos, nível de acessibilidade, modificadores e etc. Para a análise semântica é utilizado novamente as bibliotecas do compilador Roslyn. Nessa etapa é utilizado especificamente o uso da API Semântica que permite examinar a estrutura da AST, possibilitando obter, por exemplo, o tipo de uma variável de acordo com a dependência de referências assemblies, importações ou namespaces.

Ao concluir a análise semântica dos elementos da AST, a próxima etapa do processo é o Serializador RDF. O processo de serialização RDF tem como objetivo converter as relações extraídas do código-fonte e transformá-las em um conjunto de triplas (Recurso, Predicado, Objeto) no padrão RDF, seguindo o domínio de regras descrito na ontologia. O recurso e o objeto são entidades do código-fonte e o predicado é a relação definida no modelo ontológico.

Após o processo de serialização do código-fonte em triplas RDF é realizado a instância das triplas na ontologia. Essa etapa consiste na persistência das triplas RDF geradas na etapa anterior, seguindo as regras definidas na ontologia. Finalmente o gerador de domínio semântico do código-fonte realiza

o armazenamento da ontologia instanciada no repositório ontológico Apache Jena Fuseki (Jena, 2021), para que seja possível a consulta nas próximas etapas do modelo. A Figura 5.6 apresenta parte da instância da declaração do método do trecho do código-fonte 5.1 do projeto AutoMapper na ontologia.

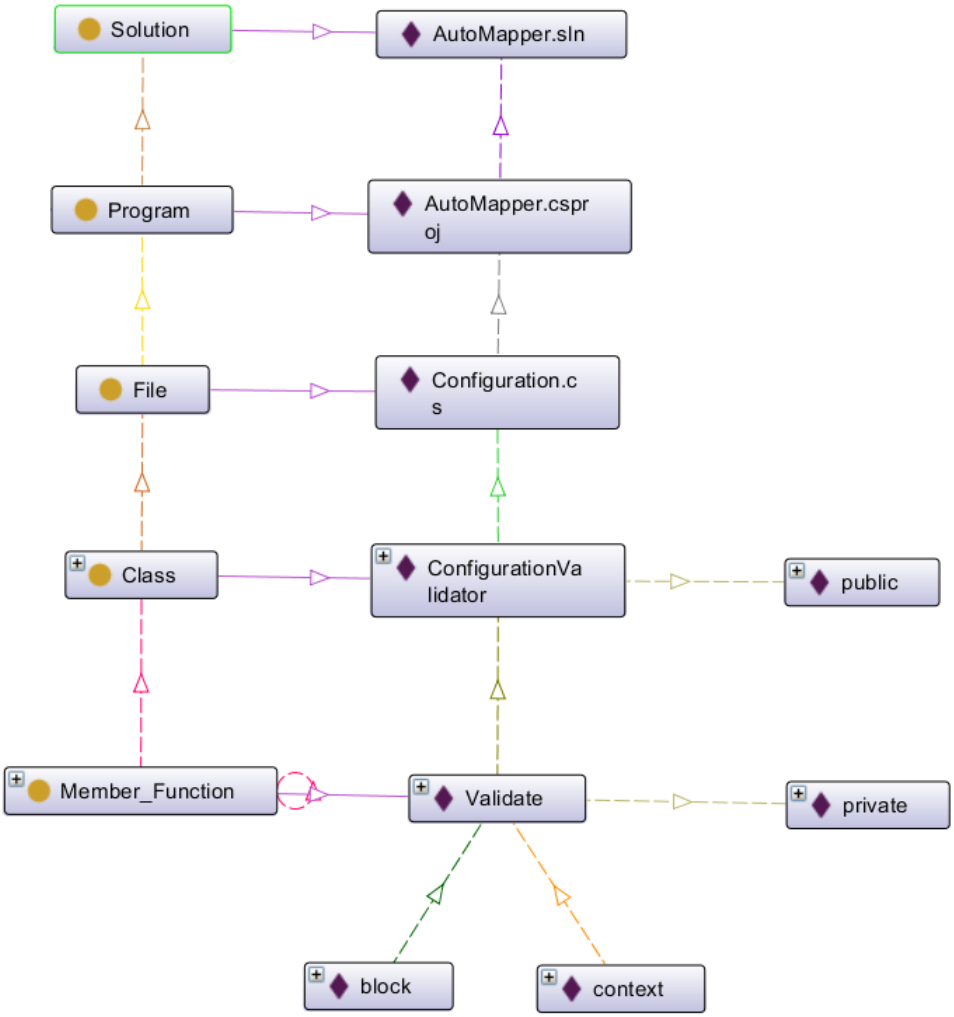


Figura 5.6: Instância na ontologia referente a declaração do método *Validate* do projeto AutoMapper (Elaborado pelo Autor).

5.3 Gerador de base de conhecimento

O módulo gerador de base de conhecimento tem como objetivo gerar uma base de conhecimento com os conceitos de domínio extraídos da ontologia do código-fonte e seus respectivos padrões. Na Figura 5.7, é apresentado a arquitetura deste módulo.

Na fase de extração de conceitos de domínio é realizado o carregamento da ontologia de código-fonte, instanciada no módulo gerador de domínio semântico do código-fonte, para a recuperação dos conceitos de domínio utilizando a linguagem SPARQL. Após a recuperação dos conceitos é realizado a decomposição dos termos de acordo com os possíveis padrões e convenções

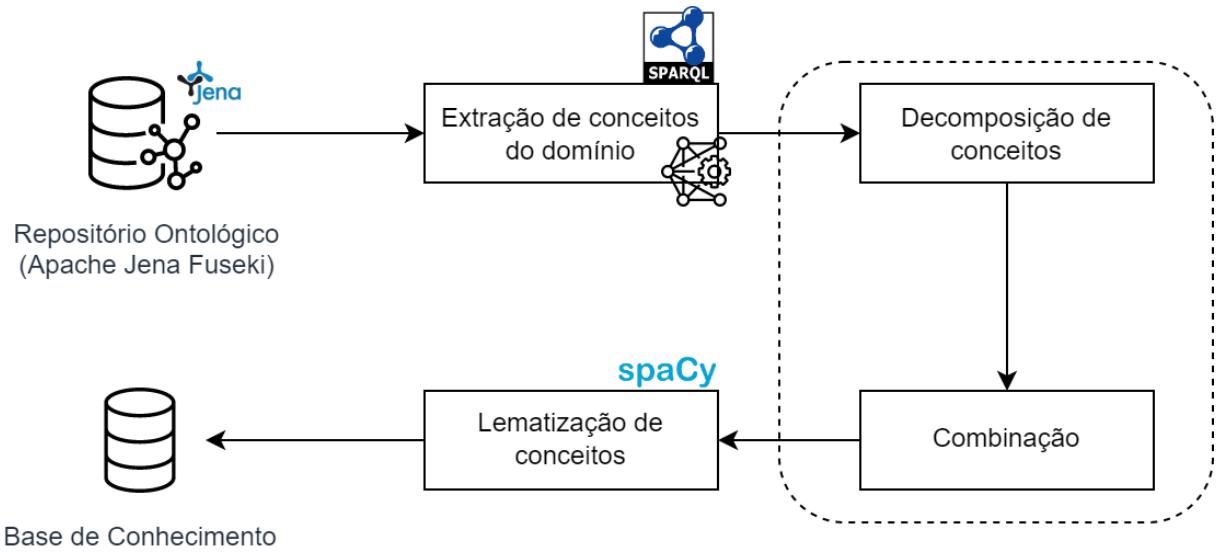


Figura 5.7: Módulo gerador de base de conhecimento (Elaborado pelo Autor).

de codificação utilizados na linguagem C Sharp (Microsoft, 2022c).

Os padrões amplamente utilizados para a nomeação de entidades são: *PascalCase*, *CamelCase* e *snake_case*. O processo de decomposição identifica a convenção de escrita do conceito e realiza a decomposição do termo. Por exemplo, para o conceito "MemberAccessQueryMapperVisitor" do tipo classe extraído do projeto AutoMapper a decomposição gera o padrão "Member Access Query Mapper Visitor".

A base de conhecimento não se limita apenas a decomposição dos conceitos, após a decomposição também é realizado um processo de combinação do termo original e do termo decomposto.

Com os processos de decomposição e combinação finalizadas para os conceitos da ontologia é realizado o processo de lematização de cada token processado na etapa anterior. Nessa etapa é utilizado a biblioteca *spaCy* para realizar o agrupamento de diferentes formas flexionadas da palavra, transformando em sua forma raiz. Por exemplo, "running" e "runs" são transformados em "run" que é a sua forma raiz.

O spaCy é uma biblioteca código aberto para Processamento Linguagem Natural em Python e projetado para criar aplicativos que processam grandes volumes de texto, podendo ser usado para construir processos de extração de informações ou sistemas de compreensão de linguagem natural (spaCy, 2021). O processo de lematização deste módulo utiliza um modelo pré-treinado no idioma Inglês, usando regras baseadas em recursos morfológicos e anotações POS (Part of Speech). O modelo é pré-treinado com textos de blogs, notícias e comentários, possuindo 685 mil chaves e 20 mil vetores exclusivos de 300 dimensões (Explosion, 2022).

Ao finalizar a etapa de decomposição, combinação e lematização, os conceitos de domínio extraídos da ontologia do código-fonte e seus respectivos padrões são persistidos em uma base de conhecimento, para o apoio na próxima etapa de reconhecimento de entidades. O código *json* 5.2 apresenta termos para o conceito "MemberAccessQueryMapperVisitor" após o processamento de todas as etapas desse módulo.

```
{
  "_id": {"$oid": "627f94b626cb5a3c0966a398" },
  "Project": 0,
  "OntologyId": "http://ooc-o/#0ACC2582AA6C6146EF24EA7EACFF7FFD10...",
  "Type": "Member_Function",
  "Name": "MemberAccessQueryMapperVisitor",
  "FilePath": "...\\QueryableExtensions\\QueryMapperVisitor.cs",
  "Pattern": [
    "memberaccess", "memberquery", "membermapper", "membervisitor",
    "memberaccessquery", "memberaccessmapper", "memberaccessvisitor",
    "memberaccessquerymapper", "memberaccessqueryvisitor",
    "memberaccessquerymappervisitor", "accessquery", "accessmapper",
    "accessvisitor", "accessquerymapper", "accessqueryvisitor",
    "accessquerymappervisitor", "querymapper", "queryvisitor",
    "querymappervisitor", "mappervisitor", "member access",
    "member query", "member mapper", "member visitor",
    "member access query", "member access mapper",
    "member access visitor", "member access query mapper",
    "member access query visitor",
    "member access query mapper visitor", "access query",
    "access mapper", "access visitor", "access query mapper",
    "access query visitor", "access query mapper visitor",
    "query mapper", "query visitor", "query mapper visitor",
    "mapper visitor"
  ]
}
```

Listing 5.2: Conceito gerado na base de conhecimento

5.4 Reconhecimento de entidades

O processo de reconhecimento de entidades inicia realizando o carregamento dos defeitos para análise e rotulagem dos conceitos. Os defeitos são extraídos utilizando a categorização dos defeitos nos repositórios dos projetos da plataforma GitHub. Na Figura 5.8 é apresentado a arquitetura deste módulo.

Ao obter os defeitos do repositório são realizadas algumas etapas pertinen-

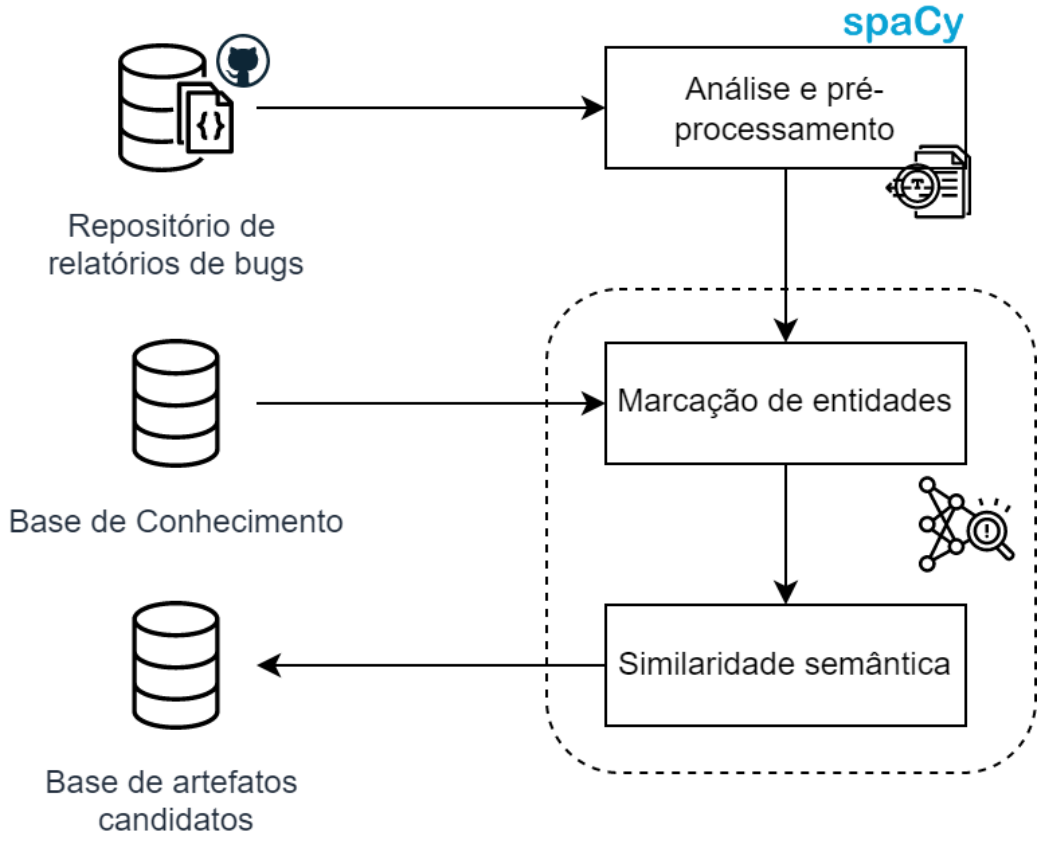


Figura 5.8: Módulo reconhecimento de entidades (Elaborado pelo Autor).

tes ao Processamento de Linguagem Natural com o apoio da biblioteca *spaCy*, com o objetivo de preparar os dados extraídos para o reconhecimento de entidades. Na etapa de análise e pré-processamento são realizadas algumas etapas antes de iniciar o reconhecimento de entidades. O pré-processamento considera o título e a descrição do relatório de defeitos. O modelo utilizado é pré-treinado no idioma Inglês com textos de blogs, notícias e comentários, possuindo 685 mil chaves e 20 mil vetores exclusivos de 300 dimensões (Explosion, 2022). As etapas de pré-processamento realizadas são:

- Segmentação de sentença: Na etapa de segmentação, o texto é dividido em sentenças, ou seja, em um parágrafo os pontos finais determinam o fim de cada sentença.
- Tokenização de palavras: Neste processo a tokenização de palavras é responsável por separar a frase em palavras ou tokens, geralmente, são considerados espaços para realizar essa separação.
- Stemização: Após a tokenização é feita a normalização dos termos encontrados normalizando as palavras na sua forma básica ou raiz. Na etapa de Stemming pode ocorrer a produção de uma palavra raiz que não possua nenhum significado.

- Lematização: Nessa etapa é feito o agrupamento de diferentes formas flexionadas da palavra, denominada Lema. A principal diferença entre Stemming e Lematização é que esta produz a palavra raiz, que tem um significado.

- Identificação de palavras de parada: Ao obter os tokens e suas derivações são realizadas as remoções de palavras e caracteres irrelevantes ou palavras vazias.

Ao finalizar o pré-processamento do título e da descrição dos relatórios de defeitos é iniciado o processo de marcação de entidades. O processo de marcação de entidades é uma etapa importante para a compreensão da linguagem natural, pois diferente da marcação de classe gramatical, é possível rotular tais entidades para o texto com informações persistidas em fontes de conhecimento estruturadas. Neste processo é possível utilizar aprendizado de máquina, onde são criados e treinados modelos para a identificação e categorização das entidades de acordo com o domínio de interesse ou a utilização de heurísticas na forma de expressões regulares Khan et al. (2016).

A etapa de marcação de entidades utiliza o Reconhecimento de Entidade Nomeada, segundo Palshikar (2013) este processo pode ser divido em duas grandes categorias:

- Genérico: nesse cenário são reconhecidas entidades que correspondam a nome de pessoas, organizações, locais, valores, datas e e-mails.

- Específico: o reconhecimento específico consiste em reconhecer entidades pertencentes a um domínio, por exemplo, nomes de proteínas e marcas de automóveis.

Nesses cenários existem três principais tipos de metodologias para o reconhecimento de entidades, que podem ter melhores precisões de acordo com o objetivo de reconhecimento, em um domínio genérico ou específico.

Para Song et al. (2018) e Palshikar (2013) o reconhecimento de entidades nomeadas possui basicamente três abordagens:

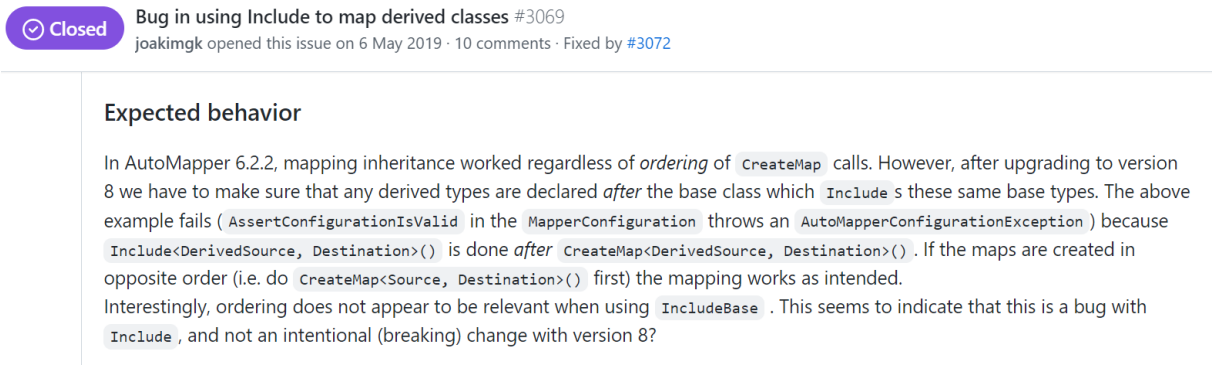
1. Reconhecimento baseado em regras: essa abordagem depende de um conjunto de regras e padrões criados manualmente por especialistas de domínio e são baseadas em conhecimentos sintáticos, linguísticos e de regras de domínio.

2. Abordagens de aprendizado de máquina: as abordagens baseadas em aprendizado de máquina não necessitam de dicionários ou conjuntos de regras para anotar as entidades em um texto, essas abordagens utilizam

principalmente os modelos como SVM (Support Vector Machine), HMM (Hidden Markov Model) e CRF (Conditional Random Fields) podendo utilizar aprendizados supervisionados e não supervisionados.

3. Dicionário de entidades: essa abordagem armazena entidades nomeadas que correspondam ao domínio em uma lista, para a utilização dessa abordagem é preciso conhecer o domínio integralmente para que sejam mantidas todas as possíveis entidades no dicionário.

No modelo de localização de defeitos do projeto LOBUS-OWL, o conhecimento de domínio do código-fonte e sua evolução são controlados, deste modo, as entidades de domínio são conhecidas em sua totalidade. Com o modelo de domínio controlado é viável a abordagem do uso de uma base de conhecimento e um mecanismo de correspondência baseado em regras para o reconhecimento de entidades, não sendo necessário o uso de técnicas avançadas de aprendizado de máquina que buscam reconhecer entidades por meio de padrões e de um grande volume de dados para o aprendizado. Na Figura 5.9 é apresentado um trecho do relatório de defeitos e sua respectiva anotação na Figura 5.10.


Closed

Bug in using Include to map derived classes #3069

joakimgk opened this issue on 6 May 2019 · 10 comments · Fixed by #3072

Expected behavior

In AutoMapper 6.2.2, mapping inheritance worked regardless of *ordering* of `CreateMap` calls. However, after upgrading to version 8 we have to make sure that any derived types are declared *after* the base class which `Include`s these same base types. The above example fails (`AssertConfigurationIsValid` in the `MapperConfiguration` throws an `AutoMapperConfigurationException`) because `Include<DerivedSource, Destination>()` is done *after* `CreateMap<DerivedSource, Destination>()`. If the maps are created in opposite order (i.e. do `CreateMap<Source, Destination>()` first) the mapping works as intended.
Interestingly, ordering does not appear to be relevant when using `IncludeBase`. This seems to indicate that this is a bug with `Include`, and not an intentional (breaking) change with version 8?

Figura 5.9: Trecho de um relatório de defeitos (Elaborado pelo Autor).

Na Figura 5.10 são destacados os conceitos identificados pelo processo de reconhecimento de entidades, de acordo com o conhecimento de domínio presente na ontologia de código-fonte do projeto. Os conceitos destacados são acompanhados do respectivo identificador na ontologia.

Neste processo podem acontecer anotações de entidades ambíguas, o processo de desambiguação dos conceitos rotulados é realizado no módulo de classificação semântica. No módulo de classificação semântica dos artefatos é considerado todas as marcações de entidades realizadas no relatório de defeitos, desta maneira é analisado o contexto presente nas demais entidades anotadas e realizada a desambiguação do termo.

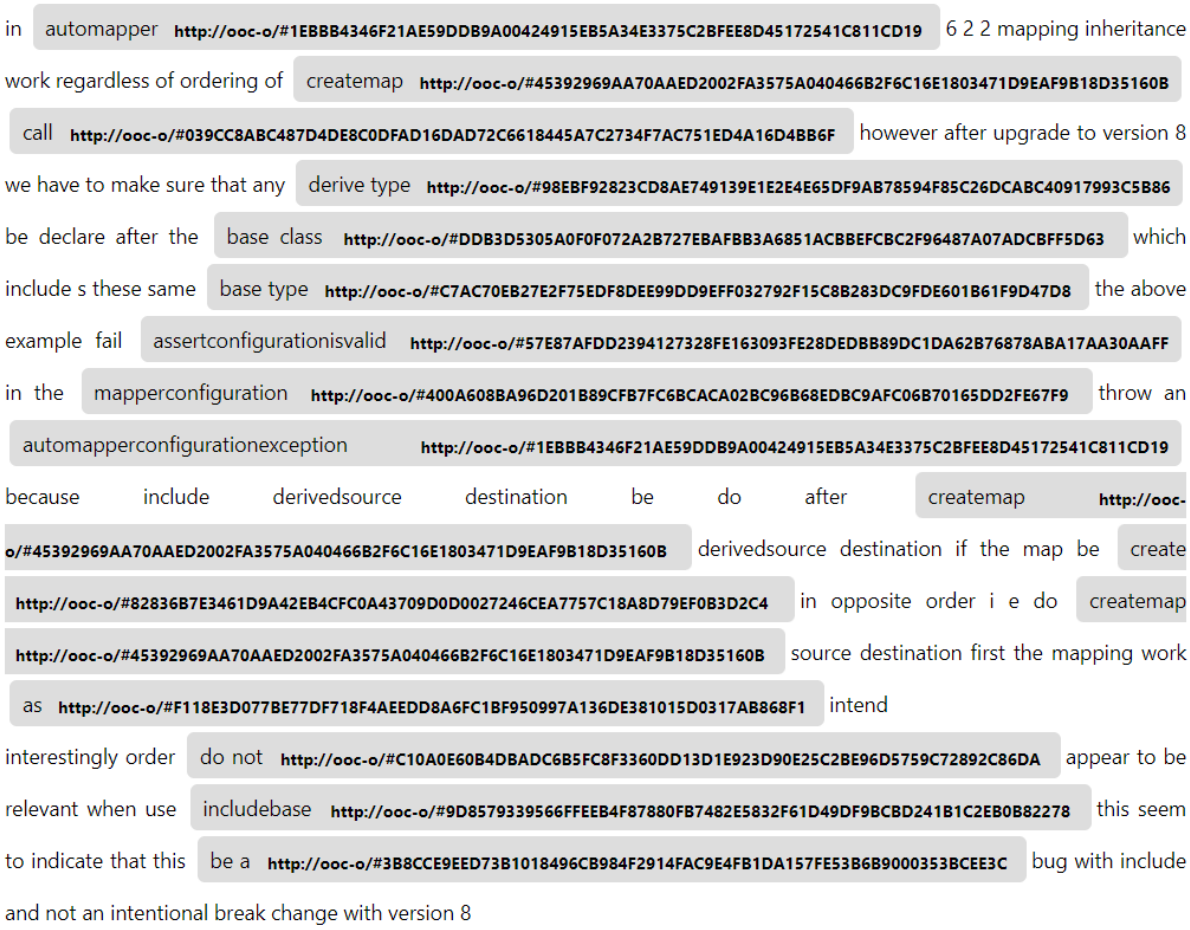


Figura 5.10: Exemplo de reconhecimento de entidades (Elaborado pelo Autor).

O componente de similaridade semântica complementa o reconhecimento de entidades verificando a similaridade dos tokens encontrados com os padrões gerados na base de conhecimento dos termos extraídos da ontologia de código-fonte. A identificação do nível de similaridade é importante para evitar ruídos no processo de reconhecimento de entidades e identificar entidades que são encontradas escritas em linguagem natural no relatório de defeitos.

Para realizar a aferição de similaridade semântica dos termos rotulados é utilizado o modelo GloVe (Pennington et al., 2014), que realiza também a incorporação de palavras e as representações vetoriais. O modelo GloVe utiliza o aprendizado não supervisionado para gerar vetores de palavras, o treinamento emprega técnicas de matrizes de coocorrência palavra-palavra e fatoração de matrizes para gerar os vetores de palavras.

Para essa etapa o modelo GloVe está pré-treinado com vetores de 300 dimensões utilizando a base de conhecimento Common Crawl (GloVe, 2022). Como os vetores estão pré-treinados, a similaridade é utilizada apenas para aferir os padrões dos conceitos decompostos que geram uma token em linguagem natural, os conceitos que são exatos do domínio não necessitam de uma

aferição de similaridade.

A aferição da similaridade semântica dos tokens identificados no relatório de defeitos considera apenas tokens que possuem um nível de similaridade maior ou igual a 80% em relação aos padrões de conceitos da base de conhecimento. Esse índice de aferição é definido de maneira empírica, com base na análise e avaliação dos conceitos rotulados nos projetos.

5.5 Classificação semântica de artefatos

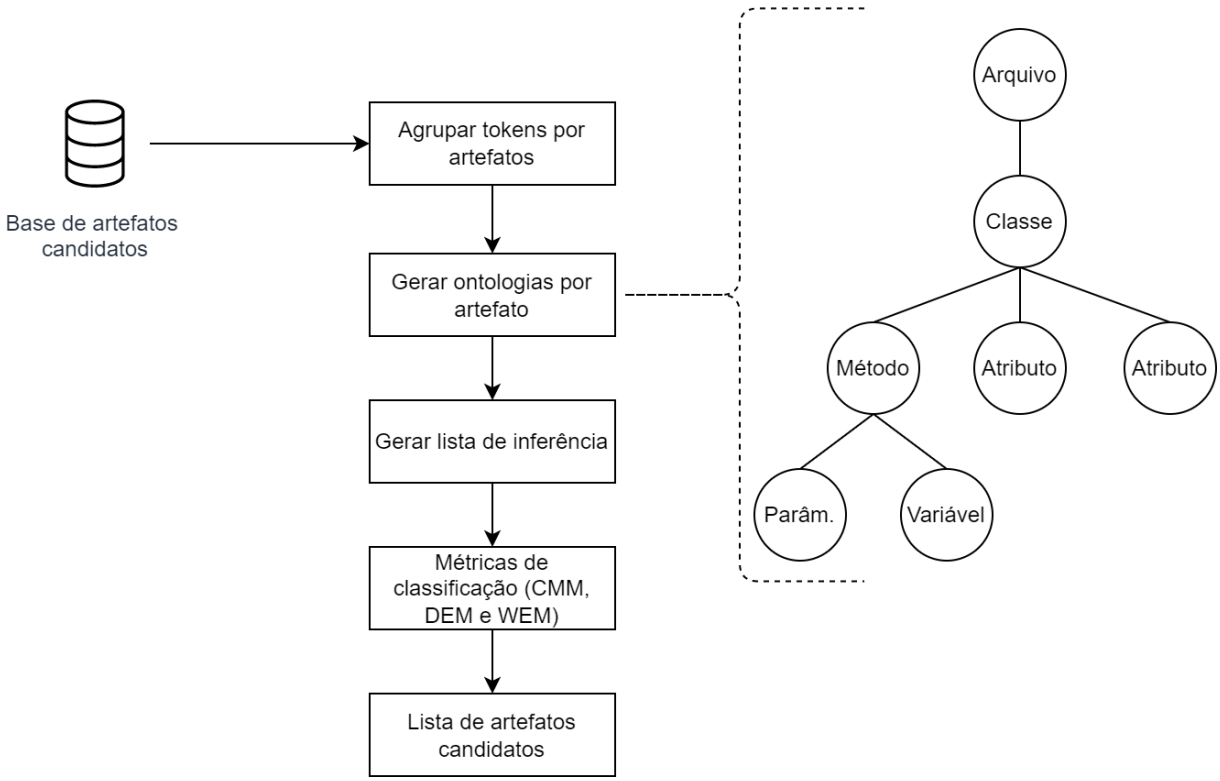


Figura 5.11: Módulo classificação semântica de artefatos (Elaborado pelo Autor).

Após a localização dos conceitos do domínio do código-fonte, reconhecidos no relatório de defeitos, é realizada a etapa de identificação e ranqueamento dos artefatos. Antes de realizar a classificação final dos artefatos é realizado um processo para gerar ontologias, para entender as relações dos conceitos encontrados no reconhecimento de entidades e garantir a desambiguação dos termos identificados no relatório de defeitos. Na Figura 5.11 é apresentada a arquitetura deste módulo.

A primeira etapa para a geração das ontologias dos conceitos identificados é agrupar os conceitos de acordo com a origem de seu artefato. Após realizar o agrupamento dos artefatos é realizada a geração de uma ontologia por artefato, considerando os conceitos identificados no módulo de reconhecimento

de entidades, o nível de aferição de similaridade semântica e as relações dos conceitos definidas na ontologia do código-fonte.

Ao concluir a geração de todas as ontologias de cada artefato encontrado, são aplicados métodos analíticos para classificar cada ontologia de acordo com sua representatividade em relação aos conceitos identificados no relatório de defeito. As métricas aplicadas avaliam diferentes aspectos representacionais em cada ontologia e definem seu ranqueamento a partir dos valores resultantes de todas as ontologias (Alani et al., 2006).

A primeira métrica considerada é a medida de correspondência de classe *Class Match Measure* (CMM) (Alani et al., 2006). A métrica CMM tem como objetivo avaliar a cobertura da ontologia com relação aos conceitos identificados no relatório de defeitos, procurando na ontologia entidades que correspondam aos conceitos identificados. Para cada conceito identificado na ontologia é considerado o nível de aferição de similaridade semântica para classificar se a entidade da ontologia corresponde a um conceito idêntico ou parcial. Os dados que estão presentes nas equações 5.1, 5.2, 5.3, 5.4 e 5.5 são: ontologia (*o*), classe (*c*) e o conjunto de classes da ontologia (*C[o]*).

$$E(o) = \sum_{c \in C[o]} SE(c) \tag{5.1}$$

$$SE(c) = \begin{cases} 1 : \textbf{\textit{se similaridade c}} = 1.0 \\ 0 : \textbf{\textit{se similaridade c}} < 1.0 \end{cases} \tag{5.2}$$

$$P(o) = \sum_{c \in C[o]} SP(c) \tag{5.3}$$

$$SP(c) = \begin{cases} 1 : \textbf{\textit{se similaridade 0.8}} \geq \textbf{\textit{c}} < 1.0 \\ 0 : \textbf{\textit{se similaridade c}} < 0.8 \end{cases} \tag{5.4}$$

$$CMM(o) = \alpha E(o) + \beta P(o) \tag{5.5}$$

Para conceitos idênticos são considerados um nível de similaridade semântica igual a 1. Os conceitos classificados como parciais consideram um nível de similaridade entre 0.8 e 0.9.

Os valores de α e β recebem respectivamente 0.6 e 0.4 para finalizar a equação, ponderando de acordo com a importância dos termos exatos ou parciais. Os valores de α e β são definidos de maneira empírica.

A próxima métrica considera a densidade da ontologia *Density Measure* (DEM). A métrica de densidade considera analisar o nível de detalhe na representação de um conceito identificado, ou seja, ao identificar uma entidade do tipo classe também pode ser encontrado um método relacionado a essa enti-

dade, atributos e variáveis. A medida de densidade tem como objetivo aferir a densidade representacional e o nível de detalhe do conhecimento da ontologia. Essa métrica é adaptada da métrica DEM (Alani et al., 2006) para o cenário de uso atual do projeto. Os dados que estão presentes nas equações 5.6 e 5.7 são: ontologia (*o*), classe (*c*), w_i pesos e os tipos de entidades da ontologia $S = \{S_1, S_2, S_3, S_4\} = \{$ *classes[o], métodos[o], atributos[o], variáveis[o]* $\}$. Os pesos foram definidos de forma empírica para classes (0,3), métodos (0,4), atributos (0,2) e variáveis (0,1).

$$dem(c) = \sum_{i=1}^{4} w_i |S_i| \tag{5.6}$$

$$DEM(o) = \frac{\sum_{i=1}^{n} dem(c)}{E(o)+P(o)} \tag{5.7}$$

A terceira métrica *Weight Measure* (WEM), tem como objetivo aferir o peso da ontologia em relação aos conceitos identificados, sem considerar as suas relações como a métrica anterior (DEM). O cálculo WEM é baseado na métrica DEM (Alani et al., 2006). Os dados que estão presentes nas equações 5.8 e 5.9 são: ontologia (*o*), classe (*c*), w_i pesos e os tipos de entidades da ontologia $S = \{S_1, S_2, S_3, S_4\} = \{$ *classes[o], métodos[o], atributos[o], variáveis[o]* $\}$. Os pesos foram definidos de forma empírica para classes (0,3), métodos (0,4), atributos (0,2) e variáveis (0,1).

$$wem(c) = \sum_{i=1}^{4} w_i |S_i| \tag{5.8}$$

$$WEM(o) = \sum_{i=1}^{n} wem(c) \tag{5.9}$$

Finalmente após calcular as métricas CMM, DEM e WEM das ontologias geradas a partir da identificação dos conceitos do relatório de defeitos e o agrupamento dos artefatos é realizado o cálculo da pontuação total. A pontuação total é calculada agregando todos os valores das métricas anteriores e determinando um peso de acordo com a importância de cada métrica para a classificação final. Os pesos para cada métrica são definidos de maneira empírica, ao final desse cálculo é definido o ranqueamento das ontologias para obter a classificação dos artefatos relevantes de acordo com os conceitos identificados no relatório de defeitos. Os dados que estão presentes na equação 5.10 são: conjunto de ontologias (*O*), ontologia (*o*) e os resultados obtidos nas métricas anteriores, sendo, $M = \{M[1], M[2], M[3]\} = \{CMM, DEM, WEM\}$. Os pesos definidos para cada um das métricas são: CMM (0,2), DEM (0,6) e WEM (0,2).

$$Score(o \in O) = \sum_{i=1}^{3} w_i \frac{M[i]}{max_{1 \leq j \leq |O|} M[j]} \tag{5.10}$$

CAPÍTULO 6

Prova de Conceito

Neste capítulo é avaliado o desempenho do projeto LOBUS-OWL detalhando o conjunto de dados utilizados para validação, as métricas para aferir o desempenho e os resultados obtidos com a execução do modelo. As métricas de avaliações são baseadas em métodos utilizados em Recuperação da Informação, conforme descrito nas subseções.

6.1 Conjunto de dados

Para avaliar a abordagem proposta neste trabalho é considerado um conjunto de dados de relatórios de defeitos de diferentes projetos de código aberto disponibilizados no GitHub. Este conjunto de relatórios de defeitos possui mais de 650 relatórios de defeitos de 6 projetos diferentes. Os relatórios de defeitos são obtidos dos repositórios dos projetos disponibilizados no GitHub e contemplam relatos de usuários variados. A Figura 6.1 apresenta a arquitetura da solução desenvolvida para obter o conjunto de dados de forma automatizada para o experimento.

O conjunto de dados contém os relatórios de defeitos extraídos de diferentes projetos de código aberto utilizando a categorização dos relatórios de defeitos dos repositórios analisados na plataforma GitHub. Para obter os relatórios de defeitos é utilizado o conjunto de APIs do GitHub para obter todas as solicitações de alteração do código-fonte do projeto e os defeitos categorizadas como *bug*, que estão vinculadas ao repositório. Após obter o conjunto de defeitos e solicitações de alterações do repositório é realizada uma heurística no texto de descrição do registro de solicitação de alteração para verificar qual o relatório de defeito que foi corrigido. Após a análise da descrição do texto de solicitação de alteração é criado um conjunto de dados que contém artefatos de código-fonte alterados na solicitação e os relatórios de defeitos vinculados

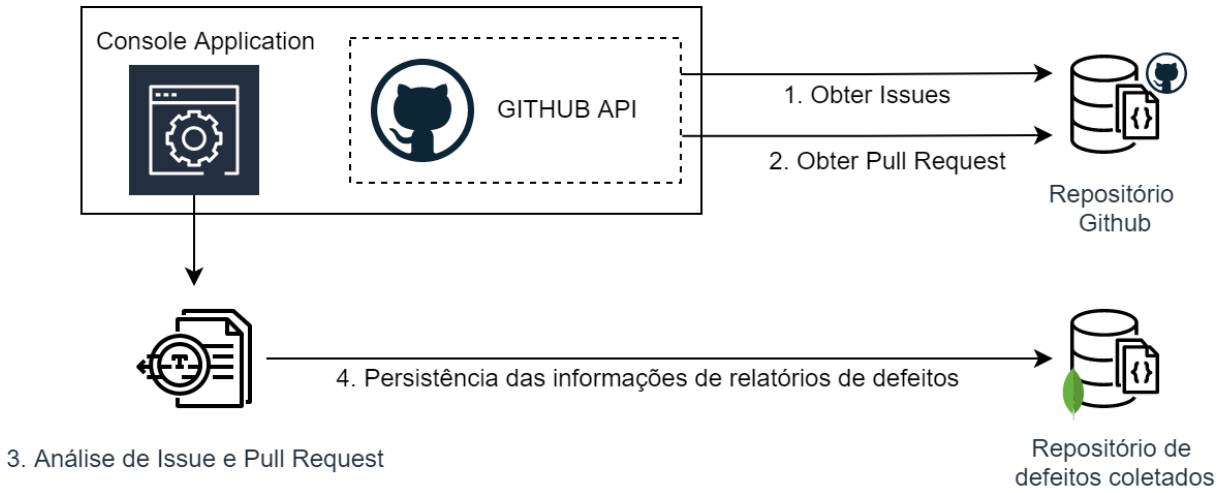


Figura 6.1: Solução para obter o conjunto de dados para o experimento (Elaborado pelo Autor).

com as manutenções realizadas.

O conjunto de dados não possui nenhum filtro para que seja validado de maneira especifica a escolha dos relatórios de defeitos para o experimento. São obtidos todos os relatórios de defeitos reportados sem distinção de descrição, considerando todas as diferentes ocorrências que o usuário da plataforma pode incluir no relatório de defeitos em um texto aberto. Os textos do relatório de defeitos podem conter imagens, trechos de código-fonte, *stack trace* e etc. O código *json* 6.1 apresenta um exemplo de registro do conjunto de dados contendo as informações obtidas do repositório do projeto AutoMapper. O registro contém os dados de solicitações de alterações e os relatórios de defeitos (*issues*) vinculados. A Figura 6.2 apresenta o diagrama dos documentos relacionados que consiste a modelagem do conjunto de dados.

```
{
  "_id":{
    "$oid":"627d3efb86445e95d8116f32"
  },
  "Project":0,
  "Number":3887,
  "Title":"Prefer derived interfaces",
  "Body":"Fixes #3886.",
  "FilesChanged":[
    "src\\AutoMapper\\Configuration\\MapperConfiguration.cs"
  ],
  "Issues":[
    {
```

```
        "Number":3886,
        "Title":"Derived interfaces are no longer mapped in 11.0.1",
        "Body":"### Source/destination types\r\n\r\n```csharp\r\n//
        Put your source/destination types here\r\npublic class Source
        \r\n{\r\n\tpublic string PropertyA {get; set;}\r\n\tpublic
        string PropertyB {get; set;}\r\n\tpublic string Name
        {get; set;} \t\r\n}\r\n\r\npublic class Target :
        IProperties\r\n{\r\n\..."
      }
   ]
}
```

Listing 6.1: Exemplo de registro do conjunto de dados para experimento

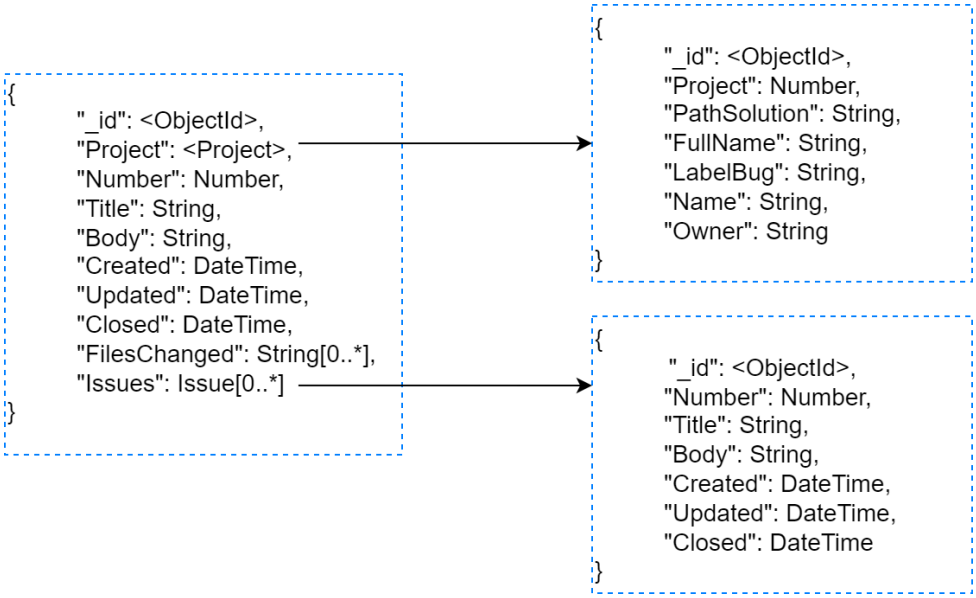


Figura 6.2: Diagrama ER dos documentos do conjunto de dados (Elaborado pelo Autor).

Para o conjunto de relatórios de defeitos são considerados apenas arquivos com a extensão ".cs", que possuem alguma modificação. Relatórios de defeitos corrigidos com arquivos de projetos de teste e arquivos que não existem na ontologia de domínio do código-fonte são desconsiderados do conjunto de dados. A geração do conjunto de dados para a avaliação da abordagem proposta considera os projetos relacionados na Tabela 6.1 e são detalhados a seguir.

Tabela 6.1: Projetos do conjunto de dados utilizados no trabalho

Projeto	**Relatórios de defeitos**
AutoMapper	82
MsBuild	72
EfCore	131
AspNetCore	156
MQTTnet	13
NLog	225

- AutoMapper: O AutoMapper é uma biblioteca construída para resolver problemas de mapeamentos entre objetos (AutoMapper, 2022a).
- MsBuild: O Microsoft Build Engine é uma plataforma para criar aplicativos, também conhecido como MSBuild. Fornece um esquema XML para um arquivo de projeto que controla como a plataforma de compilação processa e compila software (Microsoft, 2022f).
- EfCore: O EF Core é um mapeador de banco de dados de objeto (ORM) para .NET. Ele oferece suporte a consultas LINQ, controle de alterações, atualizações e migrações de esquema (Microsoft, 2022d).
- AspNetCore: O ASP.NET Core é uma estrutura de código aberto e plataforma cruzada para criar aplicativos modernos conectados à Internet baseados em nuvem, como aplicativos Web, aplicativos IoT e back-ends (Microsoft, 2022a).
- MQTTnet: MQTTnet é uma biblioteca .NET de alto desempenho para comunicação baseada em MQTT. Ele fornece um cliente MQTT e um servidor MQTT (broker) que suporta o protocolo MQTT até a versão 5 (Microsoft, 2022e).
- NLog: O NLog é uma plataforma de registro de logs para .NET com recursos avançados de gerenciamento e roteamento de logs (NLog, 2022).

6.2 Métricas de avaliação

Para a avaliação de desempenho do projeto LOBUS-OWL e a classificação da lista de artefatos candidatos para cada relatório de defeitos, são consideradas três métricas de avaliação. Para todas as métricas um valor maior indica um maior desempenho.

- Top N Rank of Files (TNRF): Essa métrica considera a posição do artefato candidato para a correção do relatório de defeitos. Os arquivos são classificados considerando as posições N (N = 1, 5, 10) da lista de artefatos

retornados. Para cada relatório de defeitos se os N principais artefatos incluir pelo menos um arquivo no qual está relacionado com a correção do defeito é considerado que o artefato está localizado (Gharibi et al., 2018).

- Mean Reciprocal Rank (MRR): Nessa métrica é medida a eficácia geral da recuperação de artefatos candidatos de um conjunto de relatórios de defeitos. A métrica classifica o inverso do resultado do conjunto de relatórios de defeitos e a classificação do primeiro artefato candidato relevante para a correção do defeito (Voorhees, 2001). Os dados que compõem a equação 6.1 são: relatórios de defeitos do projeto analisado (RD) e primeiro artefato relevante encontrado ($first_i$).

$$MRR = \frac{1}{|RD|} \sum_{i=1}^{|RD|} \frac{1}{first_i} \tag{6.1}$$

- Mean Average Precision (MAP): A métrica de precisão média é cálculo utilizado em Recuperação da Informação que avalia o desempenho de todos os arquivos candidatos relevantes para o relatório de defeitos. No processo de localização de defeitos mais de um artefato pode ser relevante para a correção do defeito, portanto essa métrica considera todas os artefatos candidatos relevantes para o relatório de defeitos (Manning et al., 2010).

$$MAP = \sum_{i=1}^{|RD|} \frac{AvgP(i)}{|RD|} \tag{6.2}$$

$$AvgP = \frac{\sum_{k \in SPrec@k}}{m} \tag{6.3}$$

$$Prec@k = \frac{\textit{posição do artefato identificado}}{\textit{posição original do artefato alterado}} \tag{6.4}$$

6.3 Resultados experimentais

Para a avaliação inicial consideramos também destacar as métricas do módulo gerador de domínio semântico do código-fonte. A Tabela 6.2 apresenta as métricas obtidas ao realizar o processamento do código-fonte dos projetos considerados no conjunto de dados dos experimentos. As ontologias são geradas considerando apenas arquivos com a extensão ".cs" e arquivos que não são relacionados a projetos de teste.

Para todos os projetos obtidos do GitHub o módulo gerador de domínio semântico do código-fonte processou com sucesso todos os arquivos do projeto, além de concluir o processamento de todas as classes, métodos, atri-

Tabela 6.2: Métricas gerador de domínio semântico do código-fonte

Projeto	**Arquivos**	**Classes**	**Métodos**	**Triplas**
AutoMapper	76	162	1.450	39.417
MsBuild	693	904	12.428	362.841
EfCore	1.704	1.950	19.576	602.378
AspNetCore	4.576	5.231	33.971	1.024.056
MQTTnet	238	243	2.081	55.020
NLog	409	501	5.627	157.029

butos, parâmetros, herança e etc., extraídos da AST e presentes nos artefatos de código-fonte. O número de triplas representam a quantidade de relações semânticas de cada projeto e incluem todas as relações presentes no código-fonte, não apenas classes e métodos.

Para a avaliação da localização de artefatos candidatos de acordo com o defeito relatado, foram utilizados os mesmos projetos aplicados no processamento do módulo gerador de domínio semântico do código-fonte que estão na Tabela 6.2. A quantidade de defeitos analisados são as mesmas apresentadas na Tabela 6.1 de acordo com o projeto.

Na Tabela 6.3 são apresentados os resultados das métricas descritas na Seção 6.2. Para a métrica TNRF é possível observar que apesar da classificação TOP@1 obter um percentual médio aproximado de 20% em relação aos 6 projetos do experimento, as classificações TOP@5 e TOP@10 atingiram ótimos resultados ao identificar artefatos candidatos relacionados aos defeitos reportados. Destaca-se que os relatórios de defeitos utilizados no experimento não foram filtrados ou classificados para a utilização do experimento, ou seja, foram considerados os relatórios de defeitos na sua forma de descrição em cenários de projetos reais. Os textos de relatórios de defeitos no GitHub podem conter imagens, trechos de código-fonte, *stack trace* ou qualquer outro elemento, pois o texto na plataforma GitHub é livre. A Figura 6.3 apresenta uma visão dos resultados de cada projeto para as métricas TOP@1, TOP@5 e TOP@10. A Figura 6.4 apresenta uma visão dos resultado para as métricas MRR e MAP.

Tabela 6.3: Desempenho de identificação e classificação de artefatos candidatos

Projeto	**TOP@1**	**TOP@5**	**TOP@10**	**MRR**	**MAP**
AutoMapper	11 (13,41%)	24 (29,27%)	34 (41,46%)	21%	19%
MsBuild	12 (16,67%)	30 (41,67%)	41 (56,94%)	29%	25%
EfCore	13 (9,92%)	32 (24,43%)	46 (35,11%)	18%	15%
AspnetCore	27 (17,31%)	51 (32,69%)	62 (39,74%)	24%	23%
MQTTnet	6 (46,15%)	9 (69,23%)	9 (69,23%)	56%	56%
NLog	30 (13,33%)	58 (25,78%)	66 (29,33%)	19%	17%

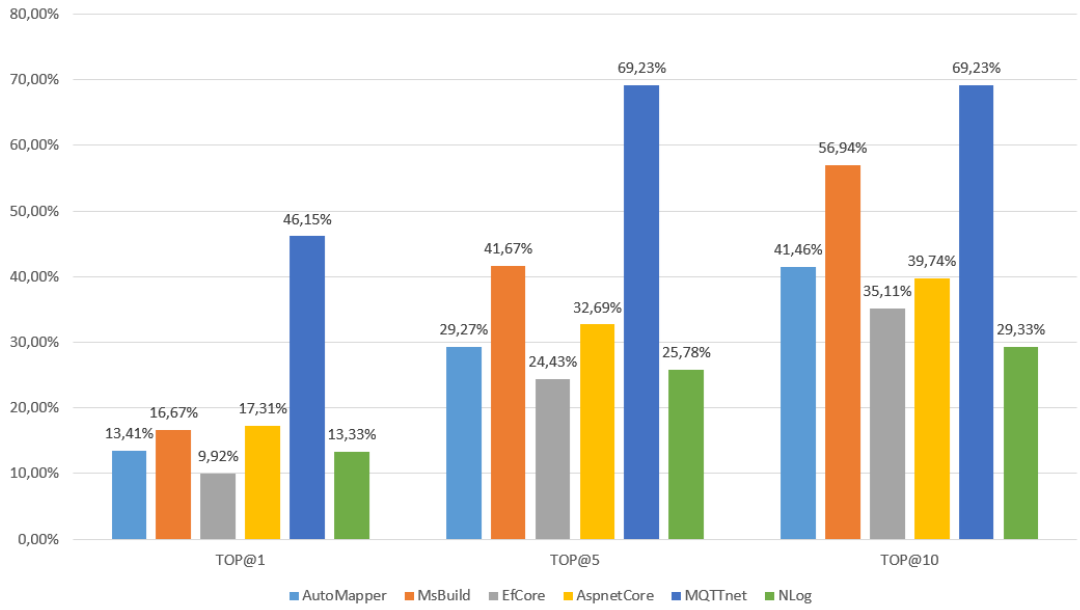


Figura 6.3: Desempenho TNRF (Elaborado pelo Autor).

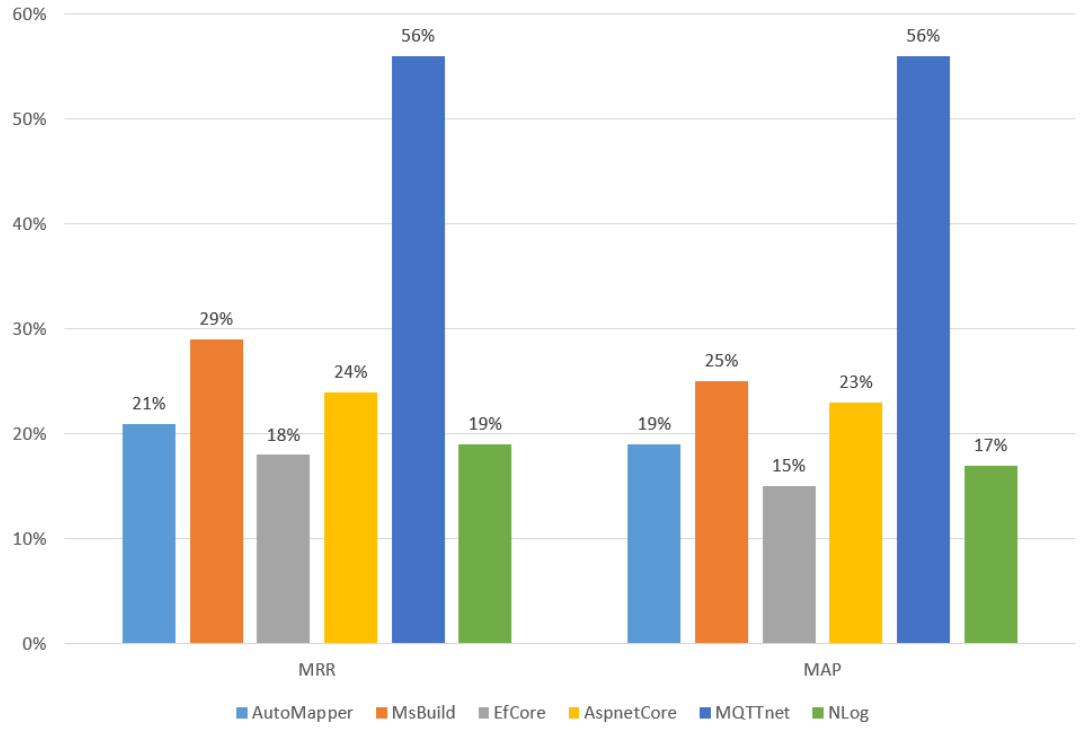


Figura 6.4: Desempenho MRR e MAP (Elaborado pelo Autor).

O projeto MQTTnet possui 13 relatórios de defeitos em seu conjunto de dados para o experimento. Mesmo com um baixo número de relatórios de defeitos o projeto foi mantido no experimento para que seja possível analisar a influência das métricas MRR e MAP para os arquivos relevantes encontrados na primeira posição.

Ambas métricas MRR e MAP consideram em suas fórmulas a posição na

classificação dos artefatos candidatos, sendo assim os resultados de MRR e MAP são influenciados também pela métrica TNRF TOP@1. Podemos observar essa relevância da classificação TOP@1 no projeto MQTTnet, onde os percentuais de classificação TOP@5 e TOP@10 são superiores ao TOP@1, porém os valores de MRR e MAP são superiores quando comparados com os demais projetos.

Apesar da relevância da métrica TOP@1, podem acontecer situações que causam distorções na análise dos resultados ao considerar apenas a métrica TOP@1 como uma avaliação relevante para o experimento. Por exemplo, quando uma correção de um relatório de defeito houver a alteração em três artefatos de código-fonte diferentes, no momento da classificação um dos artefatos foi encontrado logo na primeira posição e os demais ficaram em posições próximas ou além do TOP@10. Nesse caso as métricas de MRR e MAP podem demonstrar um resultado maior do que em um exemplo em que os três artefatos de código-fonte alterados são encontrados nas posições 3º, 4º e 5º.

As métricas MRR e MAP são relevantes, porém na análise dos experimentos podemos verificar que as métricas de TOP@5 e TOP@10 atingiram resultados expressivos para o processo de classificação de artefatos candidatos a alteração. Os resultados para os projetos MsBuild e AspnetCore apresentam resultados expressivos nas métricas TOP@5 e TOP@10. Os dois projetos são bases para o desenvolvimento de aplicações e possuem implementações que utilizam recursos avançados da linguagem, dificultando a identificação de artefatos candidatos para o relatório de defeito.

O uso da ontologia para considerar a estrutura arquitetônica e semântica do código-fonte é importante para obter artefatos que não são identificados no relatório de defeito. Esses artefatos nos projetos podem incluir herança de classes, interfaces, métodos que modificam uma implementação virtual, propriedade um elemento herdado.

Além da descoberta de artefatos que não são identificados no relatório de defeito, a estrutura semântica do código-fonte possibilita realizar a desambiguação de termos analisando o contexto das demais entidades anotadas. As ontologias geradas no módulo de classificação semântica de artefatos consideram os conceitos mais relevantes identificado de acordo com a ontologia gerada por artefato.

CAPÍTULO 7

Considerações Finais

No presente trabalho é apresentado o projeto LOBUS-OWL, um modelo para o processo de localização de defeitos apoiado por ontologia e o reconhecimento de entidades. Com o uso de ontologias é possível descrever formalmente os dados e metadados de do paradigma de programação orientado a objetos, tornando possível a localização de artefatos de código-fonte candidatos a alteração de acordo com os relatórios de defeitos reportados.

O trabalho de Aguiar et al. (2019) apresentado no Capítulo 4 expõe uma ontologia de código-fonte com suporte a linguagens de programação orientadas a objetos. A ontologia proposta por Aguiar et al. (2019) possibilita o desenvolvimento de novas pesquisas que possam explorar a recuperação da informação no domínio do código-fonte de maneira semântica. Até o trabalho de Aguiar et al. (2019) as pesquisas relacionadas ao uso de ontologias de para a representação do código-fonte careciam de uma estrutura rica em regras e relações.

As pesquisas de Gharibi et al. (2018) e Swe e Oo (2018), expostas no Capítulo 4, utilizam algoritmos para identificar a similaridade semântica e a relação entre os relatos de defeitos e os arquivos de código-fonte. Os trabalhos de Gharibi et al. (2018) e Swe e Oo (2018) apresentam lacunas que são exploradas no presente trabalho. Os dois trabalhos de Gharibi et al. (2018) e Swe e Oo (2018) fazem a análise estática do código-fonte, sem considerar as relações entre os arquivos de código-fonte. Com a utilização de ontologias para a representação paradigma de programação orientado a objetos é possível obter a interoperabilidade em diferentes linguagens de programação e fazer novas inferências no modelo semântico.

O conhecimento semântico do código-fonte possui um papel importante para no modelo de localização de defeitos proposto no presente trabalho.

A localização de defeitos com o uso de ontologias possibilita a interoperabilidade, padronização, organização e reuso da informação de domínio do código-fonte da aplicação. A modelagem da estrutura arquitetônica e semântica do código-fonte é importante para obter artefatos que não são identificados no relatório de defeito e a desambiguação de termos analisando o contexto das demais entidades anotadas.

Com o apoio de ontologias no processo de localização de defeitos é possível realizar o reconhecimento de entidades sem a necessidade de algoritmos de aprendizado de máquina profundos. Com a conclusão deste trabalho espera-se contribuir para a evolução do processo de localização de defeitos auxiliando programadores a otimizar o processo de manutenção corretiva, tornando este processo menos oneroso e mais eficiente.

A aplicação desenvolvida contribui diretamente para a evolução da pesquisa de processos de localização de defeitos em código-fonte propondo uma nova solução com a utilização de ontologia para formalização da semântica do código-fonte. A modelagem do domínio semântico com o apoio de ontologia para o processo de localização de defeito desenvolvido neste trabalho fomenta o avanço de novas pesquisas na área utilizando ontologias para a modelagem semântica.

Durante os experimentos foi observado que a falta de uma estrutura para os relatórios de defeitos do GitHub ameaça a viabilidade da localização de defeitos em alguns casos. Os textos dos relatórios de defeitos no GitHub podem conter imagens, trechos de código-fonte, *stack trace* e etc. Os trechos de código-fonte incluídos nos relatórios de defeitos podem gerar falsos positivos, causando uma identificação indevida de artefatos que não estão relacionados com o defeito.

7.1 Principais Contribuições

A seguir apresentam-se as principais contribuições que este trabalho acrescenta para a área de localização de defeitos:

- Arquitetura de um modelo para a localização de defeitos utilizando o conhecimento de domínio semântico do código-fonte com o apoio de ontologias.
- Processo de compilação e análise de projetos em C Sharp para a serialização de triplas RDF.
- Novas entidades e predicados para a ontologia OOC-O (Aguiar et al., 2019).
- Criação de uma base inédita de conhecimento semântico constituída por

seis projetos de software relevantes de código aberto (AutoMapper, MsBuild, EfCore, AspNetCore, MQTTnet e NLog).

- Conjunto de dados com base na decomposição e combinação dos conceitos instanciados na ontologia de código-fonte dos projetos.
- Processo de reconhecimento de entidades de acordo com a decomposição e combinação dos conceitos da base de conhecimento semântica do código-fonte.
- Processo para classificação semântica dos artefatos de acordo com as métricas CMM e DEM adaptadas para o modelo desenvolvido. E a métrica WEM criada para contribuir para a classificação dos artefatos.
- Processo de agrupamento e geração de ontologias por artefatos de código-fonte.
- Cálculo de ranqueamento adaptado para a classificação das ontologias dos artefatos candidatos.
- Arquitetura e projeto para obter dados de *issues* e *pull requests* de maneira automatizada utilizando o conjunto de APIs disponibilizadas pelo GitHub.
- Conjunto de dados de *issues* e *pull requests* de seis projetos de software relevantes de código aberto (AutoMapper, MsBuild, EfCore, AspNetCore, MQTTnet e NLog) disponibilizados no GitHub.

7.2 Trabalhos Futuros

Para trabalhos futuros, o modelo proposto para a localização de defeitos utilizando o reconhecimento de entidades e ontologias podem ser aprimorados utilizando outras entradas de dados para a recuperação da informação. Adicionando como parâmetro ao modelo a análise de defeitos similares torna possível a verificação de quais artefatos foram impactados no histórico de versão do código-fonte. Desse modo, é possível aprimorar a precisão da recuperação da informação.

Outras melhorias para a evolução do desempenho do projeto LOBUS-OWL podem englobar a evolução dos módulos de reconhecimento de entidades e o conhecimento do domínio da aplicação. O modulo de reconhecimento de entidades pode ser evoluído para o uso de um modelo treinado com base nas informações do domínio da aplicação e de relatórios de defeitos históricos do repositório do projeto. A evolução do conhecimento de domínio da aplicação pode ser realizado com a extensão de ontologias de acordo com o domínio

da aplicação, enriquecendo os conceitos e as relações entre as entidades do domínio do código-fonte e da aplicação.

Além disso, um avanço futuro é a disponibilização desse modelo como um *plugin* nos repositórios dos projetos do GitHub. Desta forma, ao ser relatado um novo defeito o *plugin* poderá analisar a descrição do relatório de defeito e apresentar os artefatos de código-fonte candidatos para correção. Com o plugin vinculado no repositório do código-fonte a cada novo *commit* também é possível manter a ontologia atualizada para a realização da análise dos relatórios de defeitos.

Com o apoio de ontologias também torna possível o desenvolvimento de trabalhos para a evolução de temas relacionados a detecção de *code smells*, localização de *design pattern*, reuso de componentes, localização de vulnerabilidades. Além de possibilitar a inferência de novos conhecimento a partir do domínio do código-fonte.

Referências Bibliográficas

AGUIAR, C. Z. D.; ALMEIDA FALBO, R. D.; SOUZA, V. E. S. Ooc-o: A reference ontology on object-oriented code. In: *International Conference on Conceptual Modeling*, Springer, 2019, p. 13–27.

ALANI, H.; BREWSTER, C.; SHADBOLT, N. Ranking ontologies with aktiverank. In: *International Semantic Web Conference*, Springer, 2006, p. 1–15.

ATZENI, M.; ATZORI, M. Codeontology: Rdf-ization of source code. In: *International Semantic Web Conference*, Springer, 2017, p. 20–28.

AUTOMAPPER Automapper. Online; Acessado em 05/02/2022, 2022a. Disponível em https://github.com/AutoMapper/AutoMapper

AUTOMAPPER v11 invalidoperationexception: Stack empty on mapping which used to generate a subquery 3869. Online; Acessado em 29/05/2022, 2022b. Disponível em https://github.com/AutoMapper/AutoMapper/issues/3869

BERGER OLIVIER, V. V. Helios project's bug ontology draft. Online; Acessado em 29/06/2020, 2010. Disponível em http://heliosplatform.sourceforge.net/ontologies/helios_bt.html

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The semantic web. *Scientific american*, v. 284, n. 5, p. 34–43, 2001.

BREITMAN, K. K. *Web semântica: a internet do futuro*. Grupo Gen-LTC, 2000.

BRICKLEY, D.; GUHA, R. V.; MCBRIDE, B. Rdf schema 1.1. *W3C recommendation*, v. 25, p. 2004–2014, 2014.

BUGZILLA Bugzilla. Online; Acessado em 29/05/2020, 1998. Disponível em https://www.bugzilla.org/

CHEN, D.; LI, B.; ZHOU, C.; ZHU, X. Automatically identifying bug entities and relations for bug analysis. In: *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, IEEE, 2019, p. 39–43.

DAN BRICKLEY, L. M. Foaf vocabulary specification 0.99. Online; Acessado em 01/02/2022, 2014.
Disponível em http://xmlns.com/foaf/spec

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. Elsevier Brasil, 2013.

DUCHARME, B. *Learning sparql: querying and updating with sparql 1.1*. "O'Reilly Media, Inc.", 2013.

ECLIPSE Bug 561625. Online; Acessado em 29/05/2020, 2020.
Disponível em https://bugs.eclipse.org/bugs/show_bug.cgi?id=561625

EKRAMIFARD, A.; KAHANI, M. Providing a source code security analysis model using semantic web techniques. In: *2015 International Congress on Technology, Communication and Knowledge (ICTCK)*, IEEE, 2015, p. 33–37.

EXPLOSION spacy-models. Online; Acessado em 01/05/2022, 2022.
Disponível em https://github.com/explosion/spacy-models/releases/tag/en_core_web_md-3.3.0

FERNEDA, E. *Recuperação de informação: Análise sobre a contribuição da ciência da computação para a ciência da informação*. Tese de Doutoramento, Universidade de São Paulo, 2003.

GASEVIC, D.; KAVIANI, N.; MILANOVIĆ, M. Ontologies and software engineering. 2009, p. 593–615.

GHARIBI, R.; RASEKH, A. H.; SADREDDINI, M. H.; FAKHRAHMAD, S. M. Leveraging textual properties of bug reports to localize relevant source files. *Information Processing & Management*, v. 54, n. 6, p. 1058–1076, 2018.

GITHUB Docs gihub - about issues. Online; Acessado em 15/02/2022, 2022a.
Disponível em https://docs.github.com/pt/issues/tracking-your-work-with-issues/about-issues

GITHUB Docs gihub - managing labels. Online; Acessado em 15/02/2022, 2022b.
Disponível em https://docs.github.com/pt/issues/using-labels-and-milestones-to-track-work/managing-labels

GLOVE Common crawl. Online; Acessado em 05/02/2022, 2022. Disponível em https://nlp.stanford.edu/projects/glove/

GRUBER, T. R. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, v. 43, n. 5-6, p. 907–928, 1995.

GUARINO, N.; OBERLE, D.; STAAB, S. What is an ontology? In: *Handbook on ontologies*, Springer, p. 1–17, 2009.

GUARINO, N.; ET AL. Formal ontology and information systems. In: *Proceedings of FOIS*, 1998, p. 81–97.

GUJRAL, S.; SHARMA, G.; SHARMA, S.; ET AL. Classifying bug severity using dictionary based approach. In: *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, IEEE, 2015, p. 599–602.

HAPPEL, H.-J.; SEEDORF, S. Applications of ontologies in software engineering. In: *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, Citeseer, 2006, p. 5–9.

HESSE, W. Ontologies in the software engineering process. In: *EAI*, 2005, p. 3–16.

IEEE, S. E. T. C. Ieee standard glossary of software engineering terminology. Institute of Electrical and Electronics Engineers, 1983.

ISOTANI, S.; BITTENCOURT, I. I. *Dados abertos conectados: Em busca da web do conhecimento*. Novatec Editora, 2015.

JENA, A. Apache jena fuseki. Online; Acessado em 05/09/2021, 2021. Disponível em https://jena.apache.org/documentation/fuseki2/

JEONG, G.; KIM, S.; ZIMMERMANN, T. Improving bug triage with bug tossing graphs. In: *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, p. 111–120.

KHAN, W.; DAUD, A.; NASIR, J. A.; AMJAD, T. A survey on the state-of-the-art machine learning models in the context of nlp. *Kuwait journal of Science*, v. 43, n. 4, 2016.

KIEFER, C.; BERNSTEIN, A.; TAPPOLET, J. Analyzing software with isparql. In: *Proc. rd International Workshop on Semantic Web Enabled So ware Engineering (SWESE')..(Cit. on p.)*, 2007.

LAM, A. N.; NGUYEN, A. T.; NGUYEN, H. A.; NGUYEN, T. N. Bug localization with combination of deep learning and information retrieval. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, IEEE, 2017, p. 218–229.

LASSILA, O.; SWICK, R. R.; ET AL. Resource description framework (rdf) model and syntax specification. 1998.

LOYOLA, P.; GAJANANAN, K.; SATOH, F. Bug localization by learning to rank and represent bug inducing changes. In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, p. 657–665.

MANNING, C.; RAGHAVAN, P.; SCHÜTZE, H. Introduction to information retrieval. *Natural Language Engineering*, v. 16, n. 1, p. 100–103, 2010.

MICROSOFT Roslyn. Online; Acessado em 05/09/2021, 2021.
Disponível em https://github.com/dotnet/roslyn

MICROSOFT Aspnetcore. Online; Acessado em 05/02/2022, 2022a.
Disponível em https://github.com/dotnet/aspnetcore

MICROSOFT Botframework solutions. Online; Acessado em 05/02/2022, 2022b.
Disponível em https://github.com/microsoft/botframework-solutions

MICROSOFT Csharp coding conventions. Online; Acessado em 05/02/2022, 2022c.
Disponível em https://docs.microsoft.com/pt-br/dotnet/csharp/fundamentals/coding-style/coding-conventions#naming-conventions

MICROSOFT Efcore. Online; Acessado em 05/02/2022, 2022d.
Disponível em https://github.com/dotnet/efcore

MICROSOFT Mqttnet. Online; Acessado em 05/02/2022, 2022e.
Disponível em https://github.com/dotnet/MQTTnet

MICROSOFT Msbuild. Online; Acessado em 05/02/2022, 2022f.
Disponível em https://github.com/dotnet/msbuild

MIZOGUCHI, R. Part 3: Advanced course of ontological engineering. *New Generation Computing*, v. 22, n. 2, p. 193–220, 2004.

NLOG Nlog. Online; Acessado em 05/02/2022, 2022.
Disponível em https://github.com/NLog/NLog

OMG, O. M. G. Ontology definition metamodel (odm). Online; Acessado em 29/05/2020, 2014.
Disponível em https://www.omg.org/

PALSHIKAR, G. K. Techniques for named entity recognition: a survey. In: *Bioinformatics: Concepts, Methodologies, Tools, and Applications*, IGI Global, p. 400–426, 2013.

PENNINGTON, J.; SOCHER, R.; MANNING, C. D. Glove: Global vectors for word representation. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, p. 1532–1543.

RAHMAN, M. M.; ROY, C. K. Improving ir-based bug localization with context-aware query reformulation. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, p. 621–632.

SEGUNDO, J. E. S.; CONEGLIAN, C. S. Web semântica e ontologias: um estudo sobre construção de axiomas e uso de inferências. *Informação & Informação*, v. 21, n. 2, p. 217–244, 2016.

SEGUNDO, J. E. S.; CONEGLIAN, C. S.; LUCAS, E. R. D. O. Conceitos e tecnologias da web semântica no contexto da colaboração acadêmico-científica: um estudo da plataforma vivo. *Transinformação*, v. 29, n. 3, p. 297–309, 2017.

SONG, H.-J.; JO, B.-C.; PARK, C.-Y.; KIM, J.-D.; KIM, Y.-S. Comparison of named entity recognition methodologies in biomedical documents. *Biomedical engineering online*, v. 17, n. 2, p. 1–14, 2018.

SPACY spacy. Online; Acessado em 05/10/2021, 2021.
Disponível em https://spacy.io/

STORY, H. Baetle. Online; Acessado em 29/06/2020, 2012.
Disponível em https://code.google.com/archive/p/baetle/

STUCKENSCHMIDT, H.; VAN HARMELEN, F. *Information sharing on the semantic web*. Springer Science & Business Media, 2005.

SWE, K. E. E.; OO, H. M. Bug localization approach using source code structure with different structure fields. In: *2018 IEEE 16th International Conference on Software Engineering Research, Management and Applications (SERA)*, IEEE, 2018, p. 159–164.

TRAN, H. M.; LE, S. T. Software bug ontology supporting semantic bug search on peer-to-peer networks. *New Generation Computing*, v. 32, n. 2, p. 145–162, 2014.

UDDIN, J.; GHAZALI, R.; DERIS, M. M.; NASEEM, R.; SHAH, H. A survey on bug prioritization. *Artificial Intelligence Review*, v. 47, n. 2, p. 145–180, 2017.

VOORHEES, E. M. The trec question answering track. *Natural Language Engineering*, v. 7, n. 4, p. 361–378, 2001.

W3C Web ontology language (owl). Online; Acessado em 07/05/2020, 2012.
Disponível em https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/

W3C Web ontology language (owl). Online; Acessado em 07/05/2020, 2013. Disponível em https://www.w3.org/OWL/

W3C Semantic web ontology. Online; Acessado em 07/05/2020, 2015. Disponível em https://www.w3.org/standards/semanticweb/ontology

WITTE, R.; ZHANG, Y.; RILLING, J. Empowering software maintainers with semantic web technologies. In: *European Semantic Web Conference*, Springer, 2007, p. 37–52.

ZHANG, T.; LEE, B. A bug rule based technique with feedback for classifying bug reports. In: *2011 IEEE 11th International Conference on Computer and Information Technology*, IEEE, 2011, p. 336–343.

ZIBRAN, M. F. On the effectiveness of labeled latent dirichlet allocation in automatic bug-report categorization. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2016, p. 713–715.